Script language for programmable displays

Author: Wolfgang Büscher, MKT Systemtechnik
Date : 2014-03-21 (ISO 8601)
Master file: <WoBu><ProgrammingTool>..help\scripting_01.htm

(Note: In the printable version of this file, ?/Doku/art85122_UPT_Scripting_*.pdf, external links may not work properly.
With Internet Explorer, it seems impossible to jump to the requested help topic from the script editor, so please use a *better browser*.)

Contents

See also (links to external documents, only work in HTML but not in the "easily printable" (PDF) version of this document) :

Manual for the programming tool; main index (of the programming tool's help system), feature matrix (to check if scripting is supported for a particular device/firmware), ***display interpreter*** commands, ***display interpreter*** functions .

1. Introduction

This document describes a scripting language, which has been implemented in ***some*** programmable display terminals by MKT Systemtechnik.

The script language can be used to ...

- Combine signals (i.e. generate "calculated" signals), or supervise signals received from CAN or other buses,
- implement protocols (also for CAN), which are not directly supported by the device-*firmware*, for example J1939;
- process events, which (due to their complexity) cannot be achieved in the display pages' 'Event'-definitions,
- programmatic file access, for example to implement custom specific event-logs, automatically generated error reports, etc;
- realize simple (SPS-like) flow controls, even though not for 'hard' real time (no guaranteed cycle time),
- implementation of algorithms which are too complex for the diplay's 'Event'-definition (which doesn't support loops, etc).

For most applications, you will *not* need the scripting language described in this chapter, because the functions *for which the display terminals were originally intended* can be realized without scripting. But in a few cases, the display's programmable 'Event' handlers ("page events" and "global events") will not be sufficient. This is where the scripting language can help.

The ***script language*** doesn't have anything to do with the older ***display interpreter***. This document describes the script language, not the display interpreter (the latter was used to define 'global and local events' *for the display* which were relevant for the display application). The script language is *compiled once* (not *interpreted while it runs*), using a proprietary bytecode which makes it much faster than the display interpreter, but a bit less flexible.

Throughout this document, *sourcecode* means the text which you typed into the script editor. The sourcecode will be compiled into *bytecode*. The bytecode is then executed on the target device, or in the programming tool's simulator.

Since 2011-08-02, certain 'extended' script functions may have to be unlocked (after been paid for).

1 1.1 Principle

From a user's point of view, the script is just a list of instructions and program flow control commands, entered with a simple text editor which is integrated in the programming tool. As a user, read the chapter about the script editor (integrated in the UPT programming tools), study a few examples (listed at the end of this document), and use the language reference to write your own scripts. If you are already familiar with programming languages and know what procedures, operators, arrays, strings, integer and floating point numbers are, you will only need to look at the keyword list occasionally. Otherwise, follow the hyperlinks in this document for more info ... and always remember to use your browser's "back"-button to return to the point where you started reading ;-).

After booting the programmable device, or starting the built-in simulator in the programming tool, the script will start to run in the first line of sourcecode. Typically, the first part of your script program will contain a few initialisations, like setting script variables to their default values, etc.

After that, a typical script will just sit and 'wait' for something special to happen, for example the reception of CAN messages, or if any of the programmable display pages sets a signal for the script to "do something". We'll get back to the aspects of signalling and event handling later.

2 1.2 Future Plans ..

At the current date (2013-06-05), the script language still ***lacks*** some important features which would make up a 'real' programming language, such as:

- Possibility to "break" from loops (like the "break" statement in C or PASCAL)
- Math functions like sin, cos, tan, atan, sqrt, etc
- Runtime error handling (aka exception handling, trapping of math errors, array index violations, etc)
- Dynamic memory allocation for user-defined objects of "any" size (with object sizes only known at runtime, not at compile-time)
- Functions to interact with the CAN Logger (which only exists in a few other devices)
- Multiple tasks implemented in the script language, with different priorities, semaphores, mailboxes, etc etc etc
- Speed optimisations (by the script compiler and in the runtime library)
- more rigorous error checking in the script compiler
- ...

1.3 Unlockable Features for the script language

Since 2011-08-02, only the *standard* script functions which were originally developed during the author's spare time (for a his own 'hobby' purpose) are available without an extra fee. **The *extended* functions, added to the script language *for* MKT Systemtechnik during the author's working hours *at MKT Systemtechnik*, must be unlocked** (for a moderate fee, to cover MKT's development expenses) before they can be used. These **extended script functions** include, but are not limited to, the following "unlockable" features:

- Reception and transmission of CAN messages through the script language (CAN.xyz)
- File access functions for the script language (file.xyz), which includes the serial port(s) and other objects which can be accessed "like a file".
- Script functions to communicate via TCP/IP or UDP ... planned

- other hardware-dependent, specialized functions .... planned

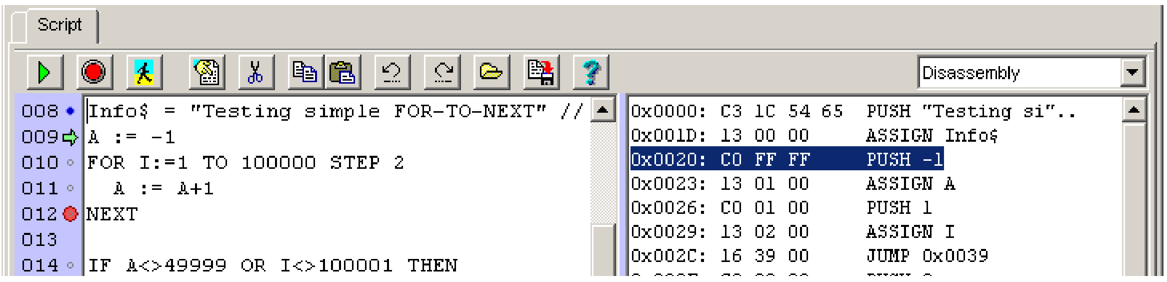
As long as these functions are not unlocked, they will simply "not work". For example, if *the script* in the programmable terminal tries to send a CAN message, the message simply won't be sent. Trying to open a file, or a serial port, will return an invalid handle.

All the above features must be unlocked for the firmware, for each device on which you want to use these features. How to request an unlock-code from the manufacturer for a particular function, in a particular device, is described in this extra document.

We apologize in advance for any inconvience caused by the unlock procedure, but the company (MKT) cannot offer all these new features for free. On the other hand, customers who don't need these functions are not forced to pay for something which they will never need.

2. Script Editor and Debugger

The script *editor* is on the 'Script' tabsheet of the programming tool. If that tab is invisible, your hardware doesn't support scripts, or the programming tool is too old.



```
Info$ = "Testing simple FOR-TO-NEXT" //
A := -1
FOR I:=1 TO 100000 STEP 2
  A := A+1
NEXT

IF A<>49999 OR I<>100001 THEN
```

```
0x0000: C3 1C 54 65   PUSH "Testing si"..
0x001D: 13 00 00      ASSIGN Info$
0x0020: C0 FF FF      PUSH -1
0x0023: 13 01 00      ASSIGN A
0x0026: C0 01 00      PUSH 1
0x0029: 13 02 00      ASSIGN I
0x002C: 16 39 00      JUMP 0x0039
```

In the script *editor*, you can use the mouse to retrieve information about a certain keyword (or, sometimes, about a variable). Just point the mouse cursor over the keyword for which you need help, and wait for two seconds (don't press any mouse button for this). If the *editor* recognizes a keyword 'under the mouse', it will show a short hint (text) near the keyword or the variable's name, data type, and current value. The hint disappears as soon as you move the mouse again.

By default, the editor uses syntax-highlighting. When enabled, the language's built-in keywords are shown in bold black characters, comments are blue, etc. Note that the syntax highlighting is not always updated while you type. In fact, the syntax highlighting function needs a valid symbol table (to identify the names of user defined procedures, variables, etc), which only exists after the program has been compiled. So after entering new sourcecode in the editor, you may have to click the 'STOP / RESET / RECOMPILE' button to update the syntax highlighting. If you find this feature too annoying, turn it off in the script editor menu (see next chapter). The editor will use plain 'black-on-white' characters then.

The size of the script *sourcecode* may be limited to 32 or 64 kByte (in some cases 256 kByte), depending on the target system. The maximum size of the bytecode (produced by the compiler) is 32 kByte on most targets. The script editor isn't aware of such target-specific limitations, unless *you* inform the programming tool about the capabilities of the target device (on the 'General Settings' tab, "Max. size of script sourcecode in kByte").
You can see the amount of source- and bytecode memory occupied by the script in the status line on the bottom of the 'Script' tab after compilation. For example, after a successful compilation, the status line will show something like:

Compilation ok, 1234 of 65536 bytecode locations used, 6 kByte sourcecode.

After compilation of the script (on the PC), it can be tested with the debugger. The debugger supports breakpoints, single-step, a disassember, the trace history, and a display for the symbol table with variable values.

To simplify the development of scripts, the editor contains several tools explained in the following chapters. These include, for example:

- the Toolbar (buttons above the sourcecode editor)
- the sourcecode editor's context menu (right-click *into the sourcecode*)
- the sidebar's context menu (right-click *into the line numbers*)

1 2.1 The Script Editor Toolbar

The toolbar contains the usual editor functions like find, copy, paste, and cut (using the windows clipboard like any other text editor). In addition, there are these graphic buttons:

RUN

Starts execution of the script, or continues execution at the last position. If the script wasn't compiled after the text was modified in the editor, it will be recompiled.
Execution may stop when the program hits a breakpoint, or an error occurrs.

STOP / RESET / RECOMPILE

Stops execution (if running), or resets / recompiles the code (if already stopped). Clicking this button with the PC's shift key held down sets the execution pointer into the line of the cursor ("caret" in the editor text).

SINGLE STEP (F7, aka 'Step In')

Executes the next command (which is marked with a green arrow on the left side of the editor text). Especially useful after hitting a breakpoint.
If the current line (marked by the green arrow) contains the call of a subroutine (procedure or function *written in the script language*), this command will ***step into*** the subroutine.

STEP OVER (F8)

Also used to single-step through the script under debugger control. In contast to the normal single step command (F7, aka 'Step In'), 'Step Over' executes a complete subroutine (procedure or function written in the script language), inclusive anything called 'from there'.

STEP OUT

Executes the rest of the current subroutine (function or procedure), until 'returning to the caller'. Typically used in combination with the 'Single Step' aka 'Step-In' command.

View "CPU" (debugger code window)

Opens the disassembly view on the right side of the script editor tab. Used for hardcore-debugging (to track down stack problems, etc).
While the disassembly-view is open, the single-step function executes one (virtual) machine code instruction per step, not one script-line !

Script editor menu

Opens a popup menu with the less frequently used 'special functions', mostly related with debugging (see one of the next chapters).
In this menu, you can also enable/disable the editor's syntax highlighting, and configure a few other script-editor related parameters.

Find text

Finds a string of characters in the script editor. The string to find is entered in the usual 'find' dialog. Then click the 'Find Next' button ("Weitersuchen" in german).

Import script from a file

Imports a script sourcecode from a plain text file. All breakpoints in the previous script will be lost.

Export script as a file

Exports the script sourcecode as a plain text file. Breakpoints will be lost when saving and re-loading the program ! Note that the script is saved as part of the terminal's display application (*.upt or *.cvt), so usually you don't need this function. The only purpose is to transfer (copy) a script from one application to another.

The left border of the script editor shows sourcecode line numbers,
indicators for 'lines with executable code', breakpoints, etc.
Possible indicators on the left side of the sourcecode editor are:

(green arrow pointing right)
Current instruction pointer .
Shows the next line to be executed. Used during single-step debugging.

(small hollow gray circle)
There is executable code in this line but the program has "not been here yet" .
Code was produced for this line when compiling, but it has not been executed yet (since the script program was started).
During debugging, you can set a breakpoint in this line by clicking on this indicator.

(blue solid circle)
"Been here since the program started".
Code was produced for this line by the compiler, and it has been executed *at least once* since the program started.
During debugging, you can set a breakpoint in this line by clicking on this indicator.
The 'Reset' function clears all 'been-here' markers (see toolbar buttons above).

(large red solid circle)
A breakpoint has been set in this line, and the program has *not* "been here" yet.
If the 'running' program hits this breakpoint, it will stop.
You can easily inspect variables then (because their values won't change while stopped).

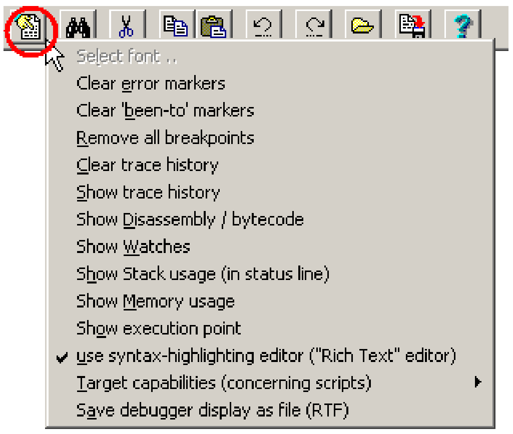
(red circle with blue center)
A breakpoint has been set in this line, and the program has 'been here' since it was last reset.

(yellow triangle with black border)
Warning or error in this line .
Something went wrong in this line, either during compilation, or during runtime. Check the error message in the status line !
To get more details about the error, point the mouse on this symbol.

Note that while editing, especially when inserting new lines in the sourcecode, the code indicators will disappear. Breakpoints remain on their fixed positions (unfortunately they cannot "move around automatically" when the sourcecode is modifed, or moved to another location).

Clicking into the sidebar (line number or symbols shown above) with the ***right*** mouse button opens the sidebar's own context menu. It contains functions like 'Show execution point', 'Show first error in line xyz', 'Toggle breakpoint in line xyz', etc. Details about the sidebar's context menu are in the next chapter.

Clicking on the 'Menü'-Button in the script editor's toolbar opens the following menu, which is mainly used while debugging:


Select font ..
Clear error markers
Clear 'been-to' markers
Remove all breakpoints
Clear trace history
Show trace history
Show Disassembly / bytecode
Show Watches
Show Stack usage (in status line)
Show Memory usage
Show execution point
✓ use syntax-highlighting editor ("Rich Text" editor)
Target capabilities (concerning scripts)
Save debugger display as file (RTF)

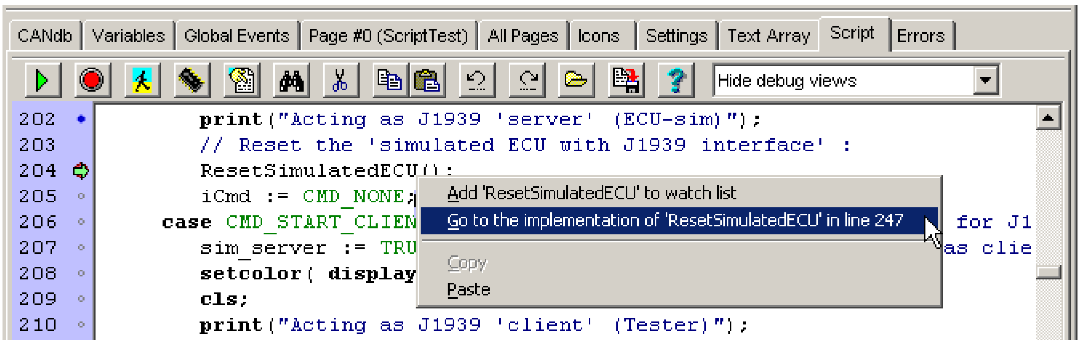
(Screenshot of the 'debug menu' on the script editor tab)

2 2.2 Hotkeys and Context Menus of the Script Editor

CTRL-C
    Copy selected text into the windows clipboard
CTRL-V
    Paste text from the windows clipboard
CTRL-F
    Find text (opens the usual non-modal 'Find' dialog)
CTRL-Z
    Undo last editing step
SHIFT-CTRL-Z
    Redo ("undo undo")
F1
    Extended help about a keyword in the script editor (sourcecode window).
    Point the mouse on a keyword, and wait until the bubble hint shows a brief information.
    If the brief information is not sufficient, press F1 (while the bubble is visible) to get *more* help, displayed in the web browser.
    Unfortunately this only works with a 'good' browser, which can jump to (scroll to) a text mark (anchor) after loading the HTML document.
    At the time of this writing (2013-08), Firefox and Iron/Chrome appeared to be 'good' browsers.
F7
    Single step (for debugging; details in the next chapter)

Furthermore, the editor supports all keyboard shortcuts implemented by Microsoft's 'Rich Text' edit control.

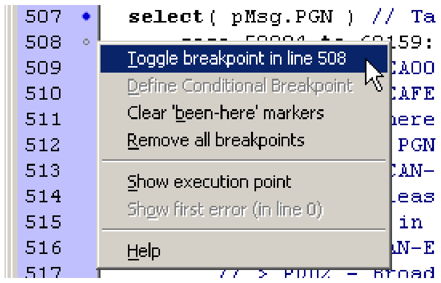
Right-click into the script editor (sourcecode) to open its context menu. For certain special functions, the 'word' in the sourcecode (under the mouse pointer) will be evaluated then, for example to add the name of a global variable as an 'expression' to the watch list, or to get help about a certain keyword, function or variable:



```
        print("Acting as J1939 'server' (ECU-sim)");
        // Reset the 'simulated ECU with J1939 interface' :
        ResetSimulatedECU();
        iCmd := CMD_NONE;
     case CMD_START_CLIEN                                   for J1
        sim_server := TRU                                  as clie
        setcolor( display
        cls;
        print("Acting as J1939 'client' (Tester)");
```

(Screenshot of the script editor with context menu, after right-click on a certain word)

Right-click into the 'Sidebar' (with line numbers and code execution indicators) opens another context menu:


Toggle breakpoint in line 508
Define Conditional Breakpoint
Clear 'been-here' markers
Remove all breakpoints
Show execution point
Show first error (in line 0)
Help

```
  select( pMsg.PGN ) // Ta
```

(Screenshot of the sidebar's context menu, opened by right-clicking on a line number)

3 2.3 Debugging

No non-trivial program will be free of errors right from the start. The programming tool has some basic debugging capabilities, listed below. To debug the code, you must run it in the programming tool. A bit of 'remote debugging' on the real target is possible via web browser (HTTP).
During a debug session, the screen *may* be split into two areas, with the sourcecode in the left half, and some other information in the right half (e.g. Bytecode, Symbole, or single Variables) .

- Breakpoints :
  Breakpoints can stop the execution of the script. To set or delete a breakpoint, click on one of the 'excutable code markers' on the left side of the editor window.
  The left mouse button simply toggles a breakpoint (on/off), the right mouse button opens a context menu with advanced options.
- Single step :
  Stop the script using the 'Stop' (or single step) button in the editor's toolbar, and continue execution step-by-step (with the single-step button) .
- Inspect variables :
  Move the mouse over the name of a global variable in the sourcecode, and wait for half a second. The program will look up the word 'under the mouse' in the list of global script variables, and show the result if it finds one. This even works while the script is running ... at least for *global* variables.
  Note: Inspecting *local* variables is not as easy as global variables, because during runtime, global variables don't have a name - just an address in the current stack frame - and thus it's extremely difficult to retrieve their values because (in contrast to global variables) their memory location cannot be found in the symbol table. Therefore, the debugger can only look up local variables if the program is currently 'pausing' in the user-defined function or procedure to which a local varible belongs. If user-defined functions & procedures call each other recursively, the debugger can only inspect local variables within the current stack frame (which is the stack frame to which the base pointer currently points).
- Stack display :
  While single-stepping, the status line of the editor may show the topmost elements on the script's stack. This function was mainly used during implementation of the script language, but you can use it to examine the stack - especially if your application makes heavy use of subroutines, and you want to find out "how the program got to the current point" (call stack). For example, after calling a subroutine with gosub Iteration in line 12, the top of the stack should contain a return address (with the code location of the next instruction after the 'gosub'). The stack display would show

  ```
  Stack[1] : CodeLine13
  ```

  which means 'there is only one item on the stack, and the item is a code address for line #13 in the sourcecode. Note that if the code address cannot be converted into a source line number, the stack display shows CodeAddr0x0ABC, which is the code location in hexadecimal form (*). If necessary, you can see the code at this address in the byte-code listing . By virtue of the debugger's stack display, it's possible to study the evaluation of RPN (reverse polish notation) in the bytecode.

  Note: The topmost element on the stack is displayed first, in the leftmost position of the

stack display line. Other elements (which have been pushed to the stack earlier) are displayed further to the right.
Other items on the stack may be ...

- Integer values: indicated by the BASIC 'integer' data type suffix (%).
- Floating point values: indicated by the BASIC 'floating point' data type suffix (!).
- String values: in double quotes (no data type suffix required)

Because the debugger / disassembler isn't aware of user defined types (not yet..), any other values on the stack are displayed as 'type X', where X is a number indicating the *data type* (not the value itself).

(*) The hexadecimal code display may be necessary if, for example, the "return address" on the stack points to the command in the same line after the "gosub". This is one more reason why you should avoid multiple commands in a single line of sourcecode - it makes debugging difficult.

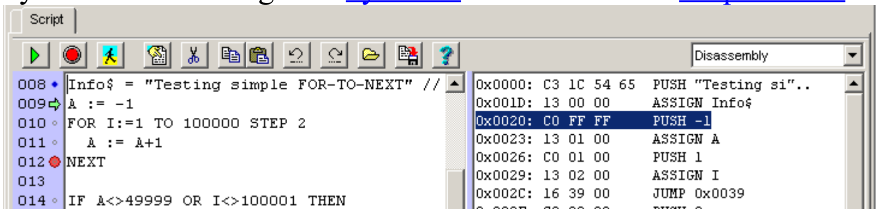
- Memory Usage Display :
  After a test session in the simulator, you should occasionally check your application's memory consumption - especially if you use a lot of strings, especially when using strings in arrays.
  To show the memory usage in the simulator / debugger, click on the script editor's menu button, and select ***Show Memory Usage*** .
  The status line will now show the memory usage (continuously updated, while the script runs), for example:

  ```
  Memory Usage : 5 of 256 stack items, peak=33; 7 of 200
  variables; 62 of 1000 data blocks, peak=85
  ```

  which means:
  5 out of 256 items on the RPN stack are currently used;
  the peak stack usage (since the script was started) was 33 out of 256 possible entries;
  7 out of a maximum of 200 global variables are used;
  62 out of a maximum of 1000 data memory blocks (with up to 64 bytes per block) are currently used;
  the peak memory usage (since the script was started) was 85 out of 1000 possible entries.
  This is a typical 'non-critical' example because all peak values are way below the maximum allowed sizes.
  If any of the three parameters (stack usage, number of global variables, or number of data blocks) gets critically close to the maximum, try to ...
  - reduce the 'stack' size by using less local variables, and less recursive procedure calls;
  - reduce the 'block' memory consumption by decreasing the array sizes;
  - reduce the 'block' memory consumption by using less strings, or assign *empty* strings to string variables if you don't need their values anymore, like:
    ```
    Info := ""; // clear this string to release its memory
    block
    ```

Note: The *Memory Usage* display on the Script tab only shows the memory *used by the script*.
This hasn't got anything to do with the memory used for the UPT's programmable *display* (for icons, display pages, etc) ! The script uses its own memory pool, so the UPT display will remain operational even if the script runs out of resources, and stops due to a programming error.
The memory usage can also be checked in the script itself, using the function system.resources .

- Disassembly display (code window) :
  Shows the bytecode in disassembled ("human readable") form. While this display is open, single-step in the debugger doesn't step through the sourcecode line-by-line, but instruction-by-instruction through the bytecode. More details in chapter 2.3.2.



```
Info$ = "Testing simple FOR-TO-NEXT" //
A := -1
FOR I:=1 TO 100000 STEP 2
  A := A+1
NEXT

IF A<>49999 OR I<>100001 THEN
```

```
0x0000: C3 1C 54 65   PUSH "Testing si"..
0x001D: 13 00 00      ASSIGN Info$
0x0020: C0 FF FF      PUSH -1
0x0023: 13 01 00      ASSIGN A
0x0026: C0 01 00      PUSH 1
0x0029: 13 02 00      ASSIGN I
0x002C: 16 39 00      JUMP 0x0039
```

- Trace History :
  Shows the last 255 events (or even more) with ...
  - CAN messages which have been *sent* (transmitted) by the device
  - CAN messages which have been *received* by the device
  - Error messages and warnings (by the device firmware or simulator)
  - Text messages and 'info lines' generated by the trace.print command

  Details about the Trace History follow in chapter 2.3.3.

- Watch Expressions :
  Shows a user-defineable selection of 'expressions' (at the moment, limited to global script variables) on the debugger panel. These can be the current values of 'simple variables', but also complete arrays and user-defined types (structs) can be inspected this way. Details about the watch list follow in chapter 2.3.5.

Hint:

In a debug session, it helps a lot to have ***two*** monitors connected to the development PC. Move the programming tool's main window to one monitor, and the LCD simulator window to the other. If you're not that lucky (only one monitor), make the size of the simulator just as large as it needs to be (to see all pixels), and use the option 'stay on top' for the LCD simulator window. You can then move the simulator into the upper right corner of your screen where it doesn't obscure any 'vital parts' of the script editor. The simulator will remain visible, even after maximizing the tool's main window, and even when the keyboard focus is not on the simulator (but inside the script editor).

See also: Debugging via Embedded Web Server (and HTML Browser)

4 2.3.1 Breakpoints and Single-Step mode

Breakpoints can stop the execution of the script. To set or delete a breakpoint, click on one of the 'excutable code markers' on the left side of the editor window.

For **Single step** operation, you can either stop the script using the 'Stop' (or single step) button in the editor's toolbar, or use a breakpoint to let it stop there.
After that, continue execution step-by-step (with the single-step button) .

In disassembly mode (see next chapter), single-step mode applies to single bytecode instructions rather than single lines of sourcecode.

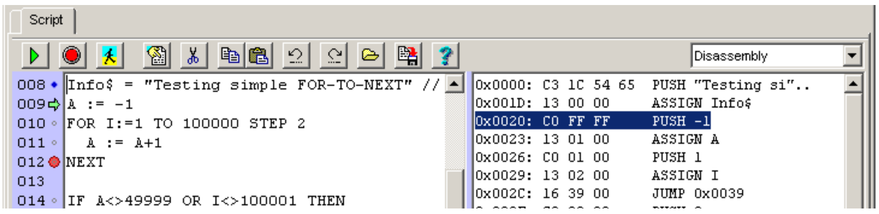
Hint:

Some devices (like MKT-View II / III, with Ethernet port and embedded web server) support debugging *without* the UPT programming tool.
That 'remote' debugger is operated via web browser (and TCP/IP, HTTP, HTML, Javascript); it also supports setting multiple breakpoints *during normal operation* and single-stepping.
To operate it, enter the device's host name or numeric IP address in the browser's address bar, followed by **/script/d** ('d' is the option for "Sourcecode Debugger").
For some stupid browsers you may have to add the transport protocol (before the address), for example: http://upt/script/d .
Details about remote debugging are here (external link).

5 2.3.2 Disassembler display (code window)

Only for advanced users !

The disassembly view can be opened through the script editor's toolbar ("chip" icon). It's an extra display panel on the right side of the editor, which shows the bytecode in disassembled ("human readable") form. While this display is open, single-step in the debugger doesn't step through the sourcecode line-by-line, but instruction-by-instruction through the bytecode. This makes it possible to see how numeric expressions ("formulas") are evaluated in the RPN (Reverse Polish Notation), and how subroutines (functions) are invoked with parameter passing via the stack.



```
Info$ = "Testing simple FOR-TO-NEXT" //
A := -1
FOR I:=1 TO 100000 STEP 2
  A := A+1
NEXT

IF A<>49999 OR I<>100001 THEN
```

```
0x0000: C3 1C 54 65   PUSH "Testing si"..
0x001D: 13 00 00      ASSIGN Info$
0x0020: C0 FF FF      PUSH -1
0x0023: 13 01 00      ASSIGN A
0x0026: C0 01 00      PUSH 1
0x0029: 13 02 00      ASSIGN I
0x002C: 16 39 00      JUMP 0x0039
```

Screenshow of script editor (left) with disassembly (right)

Immediately after the script was stopped via breakpoint, or when stepping through the bytecode in the disassembler, the current execution point (green arrow) sometimes doesn't seem to move in the sourcecode window. The resson is that each line of sourcecode usually consists of multiple bytecode instructions. Thus, to execute a single line of sourcecode, multiple single steps may be necessary (by pressing F7).

To close the disassembly window, and resume normal single step mode (line-by-line, not instruction-by-instruction), use the combo box in the upper right corner of the script editor tab, and select *Hide Debug View* instead of *Disassembly* (etc) .

6 2.3.3 Trace History

The trace history can be used any time to check the events listed below, in chronological order. It is implemented in the firmware of most devices (if the device supports scripting), but also in the programming tool (simulator). Typically, up to 255 entries can be stored in the history; more (newer) entries will overwrite the oldest part (as in a FIFO - first in, first out).
Events which *can* be recorded in the history are:

1. CAN messages which have been *sent* by the terminal ("Tx")
2. CAN messages which have been *received* by the terminal ("Rx")
3. Error messages and warnings by the system (device firmware or simulator)
4. Messages which have been "printed" into the history by the user application (script)
5. Other events, when enabled by the trace.enable flags

Chapter overview: Trace History display format, Trace History usage, Trace History invocation .

6.1 2.3.3.1 Trace History display format

The display format of CAN messages in the Trace History is half-way compatible with Vector's widespread "ASII" format for CAN logfiles. Thus, even for devices without an integrated CAN logger / snooper, it is possible to trace CAN-bus related problems (which is especially helpful when implementing 'exotic' CAN protocols in the script language).

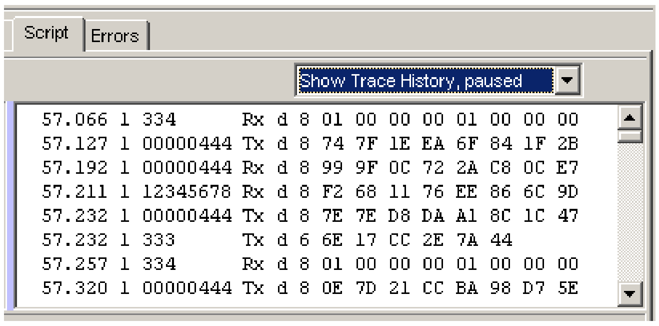
CAN message format in the Trace History:

Timestamp	Bus	CAN-ID	Rx/ Tx	d	Length	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
57.211	1	12345678	Rx	d	8	F2	68	11	76	EE	86	6C	9D

CAN-Identifiers with 11 bit (standard frames) are displayed with 3 digits. 29-Bit-Identifier (extended frames) are shown with 8 digits as in the example.
Messages (text lines) entered into the history by command ('trace.print') can have any format; only the timestamp (in seconds, with three decimal places) are added automatically at the begin of each line. The second counter starts at zero when the device is turned on, or when the simulator is started / reset.

6.2 2.3.3.2 Trace History usage

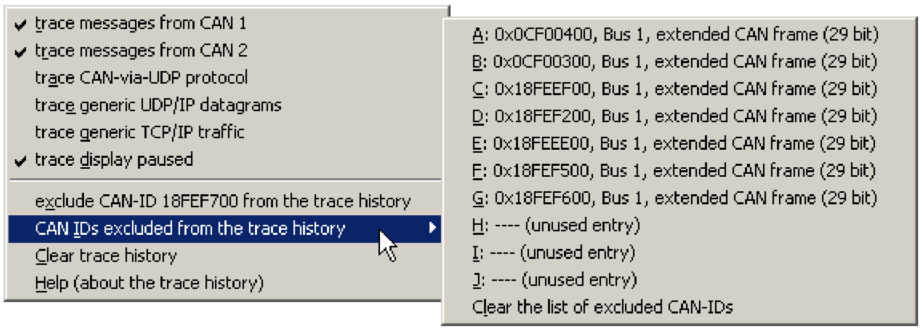
In the simulator (integrated in the programming tool), the Trace History can be displayed on the right side of the 'Script' tab. To achieve this, select 'Show Trace History' in a combo box on the script toolbar:



```
57.066 1 334      Rx d 8 01 00 00 00 01 00 00 00
57.127 1 00000444 Tx d 8 74 7F 1E EA 6F 84 1F 2B
57.192 1 00000444 Rx d 8 99 9F 0C 72 2A C8 0C E7
57.211 1 12345678 Rx d 8 F2 68 11 76 EE 86 6C 9D
57.232 1 00000444 Tx d 8 7E 7E D8 DA A1 8C 1C 47
57.232 1 333      Tx d 6 6E 17 CC 2E 7A 44
57.257 1 334      Rx d 8 01 00 00 00 01 00 00 00
57.320 1 00000444 Tx d 8 0E 7D 21 CC BA 98 D7 5E
```

(Trace History displayed in the programming tool)

After selecting 'Show Trace History, paused' you can scroll back through the history, as long as the limited trace memory permits.
Switching to 'Show Trace History, running' will permanently append new entries at the end of the display, and the vertical scroller will automatically be moved to the end of the list, so the newest entry is always visible.
Right-click into the Trace History (in the programming tool) opens the following context menu:


✓ trace messages from CAN 1
✓ trace messages from CAN 2
trace CAN-via-UDP protocol
trace generic UDP/IP datagrams
trace generic TCP/IP traffic
✓ trace display paused
exclude CAN-ID 18FEF700 from the trace history
CAN IDs excluded from the trace history
Clear trace history
Help (about the trace history)
A: 0x0CF00400, Bus 1, extended CAN frame (29 bit)
B: 0x0CF00300, Bus 1, extended CAN frame (29 bit)
C: 0x18FEEF00, Bus 1, extended CAN frame (29 bit)
D: 0x18FEF200, Bus 1, extended CAN frame (29 bit)
E: 0x18FEEE00, Bus 1, extended CAN frame (29 bit)
F: 0x18FEF500, Bus 1, extended CAN frame (29 bit)
G: 0x18FEF600, Bus 1, extended CAN frame (29 bit)
H: ---- (unused entry)
I: ---- (unused entry)
J: ---- (unused entry)
Clear the list of excluded CAN-IDs

(Context menu to control the Trace History in the programming tool)

The menu shown above can be used to suppress certain CAN message identifiers in the Trace History. This feature is often used to avoid 'flooding' the display with non-important, but frequently transmitted CAN frames.

On a real target device, the Trace History can be accessed (inspected) through the system menu. Select 'Diagnostics' .. 'TRACE History' there. Details about the system menu are in document #85115 (System Menu and Setup in programmable CAN display terminals by MKT).

See also: 'Trace History invocation' in *this* document.

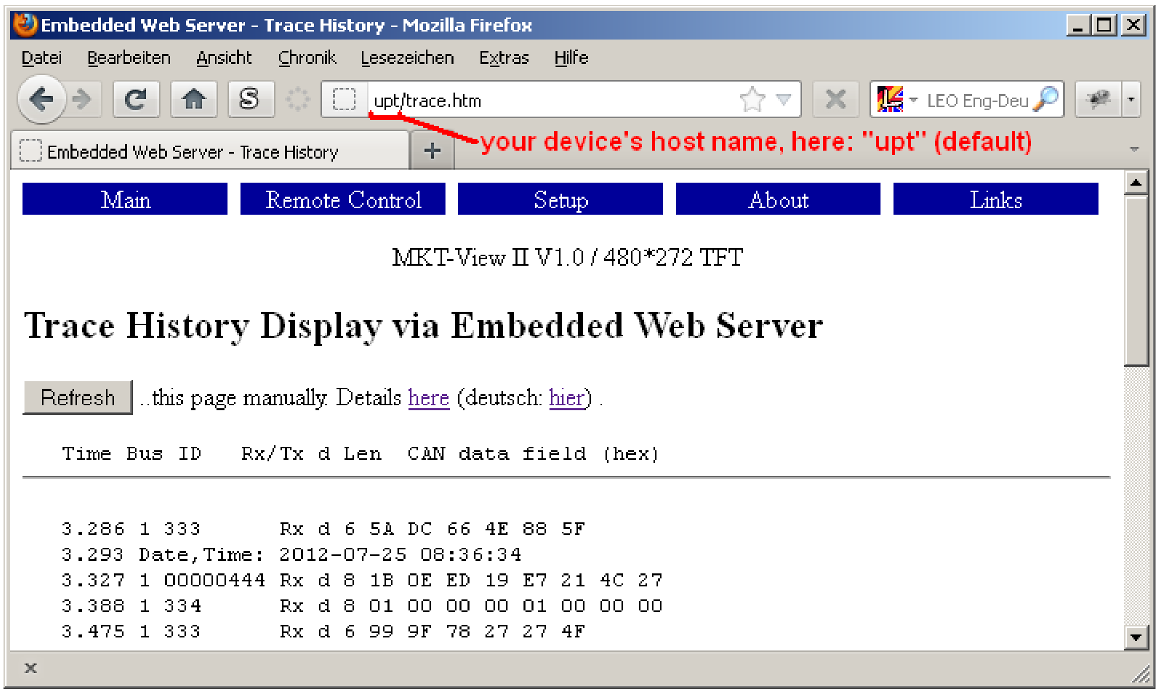
6.3 2.3.3.3 Excluding certain CAN message identifiers from the Trace History

To suppress ("blacklist") a certain CAN message identifier for the display, click on its hexadecimal identifier with the *right* mouse button in the trace display, and select 'Exclude CAN-ID from the trace history'.
Alternatively items in the blacklist can be edited (in hexadecimal form) via context menu, sub menu titled 'CAN IDs excluded from the trace history'. This menu also shows a complete list of all currently black-listed CAN message identifiers.
Suppressing certain CAN identifiers as described above only affects *the trace history display in the programming tool (simulator)*, but not the *'real' device* (i.e. hardware like MKT-View III which also has a built-in trace history).

In a 'real' device (but also in the simulator), *the script itself* can access the CAN-ID-blacklist via trace.can_blacklist[i] .

6.4 2.3.3.4 Accessing the Trace History via web browser

The easiest method to check the Trace History *in a real device* is through your web browser (for all devices with Ethernet and embedded web server). In most cases, you can access the device easily through its host name (which is "upt" by default, but the name may have been changed in the device's network setup). The full URL would be "http://upt/trace.htm", but the protocol name (http) is usually added by the web browser internally, and not shown in the address bar. Here for example the Trace History displayed in the author's favourite browser:


Embedded Web Server - Trace History - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
upt/trace.htm
LEO Eng-Deu
Embedded Web Server - Trace History
your device's host name, here: "upt" (default)
Main
Remote Control
Setup
About
Links
Refresh

MKT-View II V1.0 / 480*272 TFT

Trace History Display via Embedded Web Server

..this page manually. Details here (deutsch: hier) .

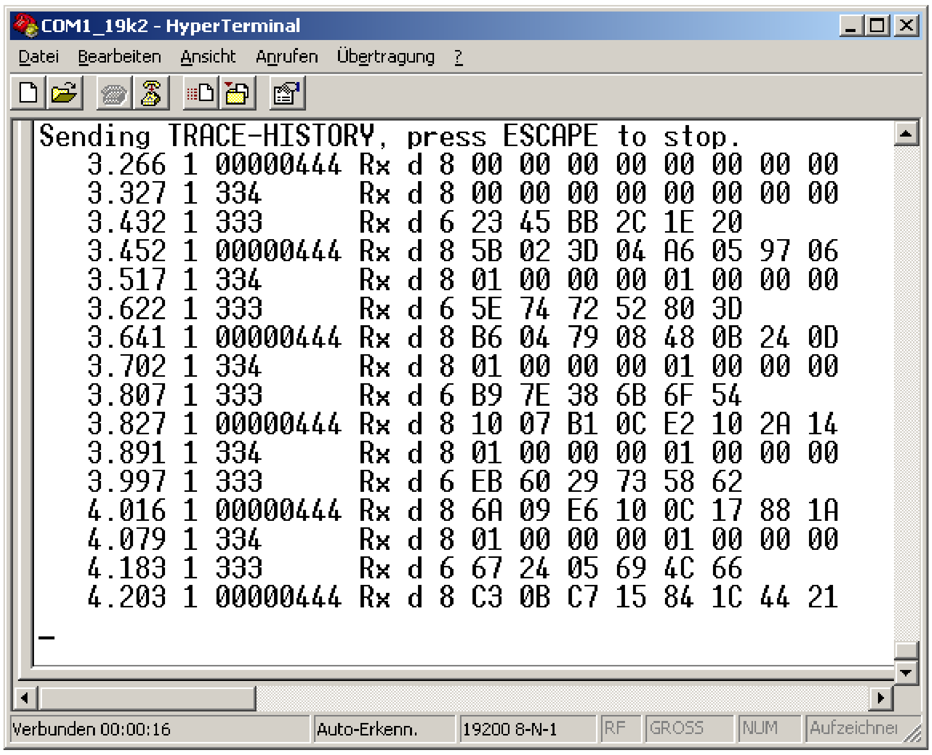
```
   Time Bus ID    Rx/Tx d Len  CAN data field (hex)

   3.286 1 333       Rx d 6 5A DC 66 4E 88 5F
   3.293 Date,Time: 2012-07-25 08:36:34
   3.327 1 00000444 Rx d 8 1B 0E ED 19 E7 21 4C 27
   3.388 1 334       Rx d 8 01 00 00 00 01 00 00 00
   3.475 1 333       Rx d 6 99 9F 78 27 27 4F
```

(Trace History read via embedded web server, and displayed in a web browser)

In case of problems with the network connection (or the web browser), see 'Troubleshooting' in the description of the embedded web server.

6.5 2.3.3.5 Reading the Trace History via serial interface

Alternatively, and depending on the hardware, the trace history can be read as plain text through a serial port, and saved as a text file on the PC.
To read the trace history via serial interface (RS-232 or Virtual COM Port), use a terminal program like 'Hyperterminal', and enter the command *****trace***** followed by carriage return ("Enter" key). The default baudrate for the serial interface is 19200 bits/second for most devices with a 'real' RS-232 port (like MKT-View 2). For device which only have a Virtual COM Port (looks like an USB device adapter from outside), try 115 kBit/second.
Since the serial port might have been reconfigured by the application (script), theres no easy way to tell the correct communication parameters here. Usually either "115200 8-N-1" or "19200 8-N-1" should work.


COM1_19k2 - HyperTerminal
Datei Bearbeiten Ansicht Anrufen Übertragung ?
Verbunden 00:00:16
Auto-Erkenn.
19200 8-N-1
RF
GROSS
NUM
Aufzeichnen

```
Sending TRACE-HISTORY, press ESCAPE to stop.
    3.266 1 00000444 Rx d 8 00 00 00 00 00 00 00 00
    3.327 1 334      Rx d 8 00 00 00 00 00 00 00 00
    3.432 1 333      Rx d 6 23 45 BB 2C 1E 20
    3.452 1 00000444 Rx d 8 5B 02 3D 04 A6 05 97 06
    3.517 1 334      Rx d 8 01 00 00 00 01 00 00 00
    3.622 1 333      Rx d 6 5E 74 72 52 80 3D
    3.641 1 00000444 Rx d 8 B6 04 79 08 48 0B 24 0D
    3.702 1 334      Rx d 8 01 00 00 00 01 00 00 00
    3.807 1 333      Rx d 6 B9 7E 38 6B 6F 54
    3.827 1 00000444 Rx d 8 10 07 B1 0C E2 10 2A 14
    3.891 1 334      Rx d 8 01 00 00 00 01 00 00 00
    3.997 1 333      Rx d 6 EB 60 29 73 58 62
    4.016 1 00000444 Rx d 8 6A 09 E6 10 0C 17 88 1A
    4.079 1 334      Rx d 8 01 00 00 00 01 00 00 00
    4.183 1 333      Rx d 6 67 24 05 69 4C 66
    4.203 1 00000444 Rx d 8 C3 0B C7 15 84 1C 44 21
_
```

(Trace History read via serial port, and displayed in 'HyperTerminal')

Unfortunately, in Windows 'Vista' and Windows 7, HyperTerminal isn't included anymore. In this case, we recommend using PuTTY (freeware terminal software) to read the Trace History through the serial port (if memory card or Ethernet is not available).

6.6 2.3.3.6 Saving the Trace History as a file

Last not least, if your PC (or your local network) refuses to establish a TCP/IP connection to the terminal, you can alternatively retrieve the Trace History as a plain text file by saving it on the memory card: First invoke the trace history on the device's own screen, press the 'Menu' button (softkey) there, and select 'Save Trace as file'. The firmware will dump the trace memory as plain text files, beginning with the name 'TRACE000.TXT'. With each new call of the 'Save as file'-function, a new file will be written (TRACE001.TXT, TRACE002.TXT, and so on).

The same function can be invoked directly from the script (trace.save_as_file).

The Trace History (in RAM) will be deleted when turning off the device, because it is only buffered in RAM for performance reasons. It cannot (and shall not) replace the CAN logger / 'Snooper' which is integrated in certain devices.

The trace accumulated *in the simulator* (i.e. the programming tool) can be copied into an own document (and thus be saved 'as a file' on the PC) as follows:

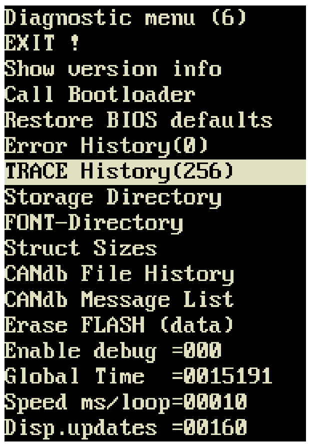
1. Set the focus into the trace history display (via mouse click, etc)
2. press CTRL-A (select all) or mark the interesting part of the trace via mouse
3. press CTRL-C to copy the selected text into the windows clipboard (as usual)

4. set the focus into your own document, and press CTRL-V to paste (insert) the text from the clipboard

6.7 2.3.3.7 Trace History invocation

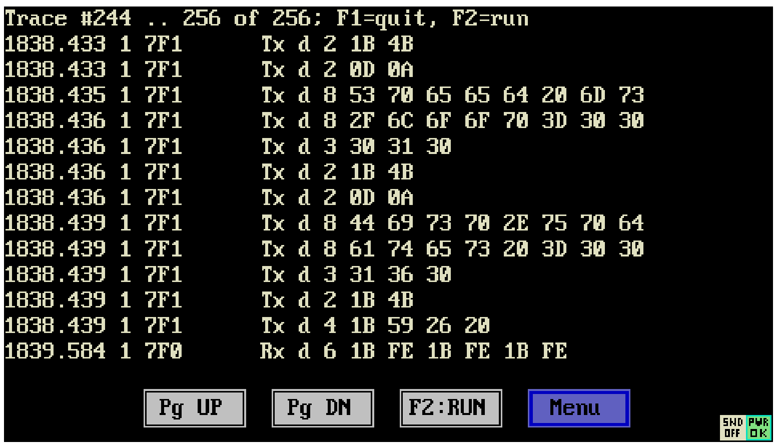
There are several methods to invoke the Trace History Display locally, i.e. show it on the terminal's own screen.
Here, for example, how to invoke the Trace History Display in the MKT-View II / MKT-View III :

1. Draw the gesture 'U' on the touchscreen to enter the device's shutdown / system popup window.
   Alternatively (for devices without a touchscreen), press F2 + F3 simultaneously to enter the system menu.
2. Select 'SETUP' if not already there.
3. In the 'Main system setup', select 'DIAGNOSTICS'.
4. Select 'Trace History'. The number in parenthesis (after the menu item) shows the number of entries which are currently stored in the Trace History.



```
Diagnostic menu (6)
EXIT !
Show version info
Call Bootloader
Restore BIOS defaults
Error History(0)
TRACE History(256)
Storage Directory
FONT-Directory
Struct Sizes
CANdb File History
CANdb Message List
Erase FLASH (data)
Enable debug =000
Global Time  =0015191
Speed ms/loop=00010
Disp.updates =00160
```

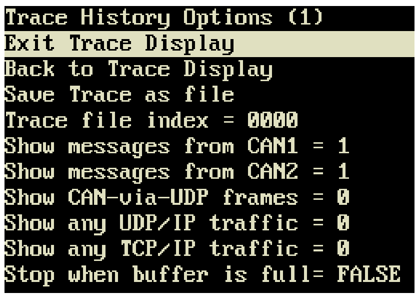
---->



```
Trace #244 .. 256 of 256; F1=quit, F2=run
1838.433 1 7F1      Tx d 2 1B 4B
1838.433 1 7F1      Tx d 2 0D 0A
1838.435 1 7F1      Tx d 8 53 70 65 65 64 20 6D 73
1838.436 1 7F1      Tx d 8 2F 6C 6F 6F 70 3D 30 30
1838.436 1 7F1      Tx d 3 30 31 30
1838.436 1 7F1      Tx d 2 1B 4B
1838.436 1 7F1      Tx d 2 0D 0A
1838.439 1 7F1      Tx d 8 44 69 73 70 2E 75 70 64
1838.439 1 7F1      Tx d 8 61 74 65 73 20 3D 30 30
1838.439 1 7F1      Tx d 3 31 36 30
1838.439 1 7F1      Tx d 2 1B 4B
1838.439 1 7F1      Tx d 4 1B 59 26 20
1839.584 1 7F0      Rx d 6 1B FE 1B FE 1B FE
```

(Invocation of the Trace History via system menu, and display on the device's "local" screen)

The softkey 'Menu' (marked in blue in the above screenshot) can be used to open the following menu:



```
Trace History Options (1)
Exit Trace Display
Back to Trace Display
Save Trace as file
Trace file index = 0000
Show messages from CAN1 = 1
Show messages from CAN2 = 1
Show CAN-via-UDP frames = 0
Show any UDP/IP traffic = 0
Show any TCP/IP traffic = 0
Stop when buffer is full= FALSE
```

(Menu with Trace History Options, here on an MKT-View III)

In *most* (but not all) devices by MKT, the following entries are available in the menu shown above:

`Exit Trace Display`

Leave the trace history display, and return to the caller (which is usually the system menu)

`Back to Trace Display`

Leave the 'Options' menu, and switch back to the trace history display

`Save Trace as file`

Saves the trace history buffer as a plain text file on the SD memory card. Details here.

`Show messages from CAN1 = {0,1}`

1 (Default): Show CAN-messages sent to, and received from, the first CAN interface.
0: Don't show these messages (and don't enter them into the history buffer, from now on).

`Show messages from CAN2 = {0,1}`

1 (Default): Show CAN-messages sent to, and received from, the second CAN interface. 0: Don't show these messages.

`Show messages from CAN-via-UDP = {0,1}`

1: Show CAN-messages 'tunneled' via UDP (Ethernet). 0 (Default): Don't show these messages.

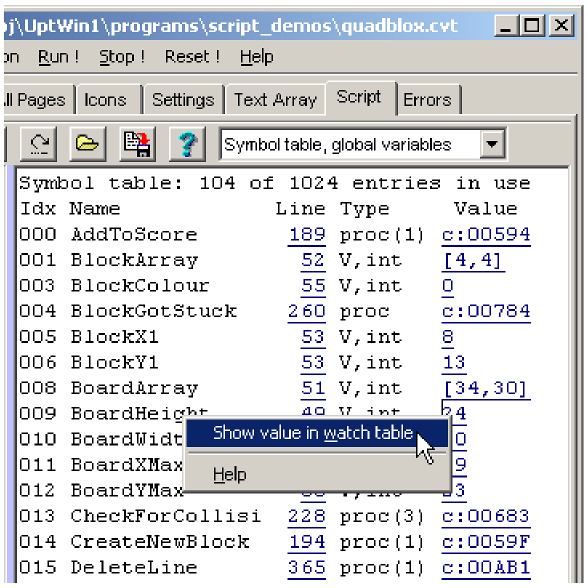
`Stop when buffer is full`

Special option to trace problems during startup / network boot / initialisation.
TRUE : The trace history will be stopped when the history buffer is full. Thus only the 'oldest' entries are available.
FALSE: The trace history will continue running (even when the buffer is full), thus only the 'newest' entries are available.

The default setting is 'FALSE', i.e. the trace historie will not be stopped (at least not automatically), and (depending on the firmware) only the last 255 or 511 entries can be viewed or exported.

7 2.3.4 Symbol Table with variable display

In the programming tool, the symbol table can be displayed on the right side of the main window. Select the item 'Symbol table, complete' or 'Symbol table, global variables' in the combo box in the scipt editor's toolbar. The 'complete' symbol table also shows the names and locations of local variables (which cannot be inspected). The display with 'global' variables only shows global symbols (global variables of the script, functions, procedures, and constants).
In the tabular display, all symbols are sorted by name, which makes this display a valuable tool for 'navigation': Click on the 'Line number' (which is shown like a hyperlink) to scroll the script editor to the line in which a variable (or function, procedure, constant, etc) is defined.


j\UptWin1\programs\script_demos\quadblox.cvt
on Run ! Stop ! Reset ! Help
ll Pages Icons Settings Text Array Script Errors
Symbol table, global variables
Show value in watch table
Help

```
Symbol table: 104 of 1024 entries in use
Idx Name              Line Type     Value
000 AddToScore         189 proc(1)  c:00594
001 BlockArray          52 V,int    [4,4]
003 BlockColour         55 V,int    0
004 BlockGotStuck      260 proc     c:00784
005 BlockX1             53 V,int    8
006 BlockY1             53 V,int    13
008 BoardArray          51 V,int    [34,30]
009 BoardHeight         49 V,int    24
010 BoardWidt                       0
011 BoardXMax                       9
012 BoardYMax                       3
013 CheckForCollisi    228 proc(3)  c:00683
014 CreateNewBlock     194 proc(1)  c:0059F
015 DeleteLine         365 proc(1)  c:00AB1
```

Screenshot of the script symbol table in the programming tool

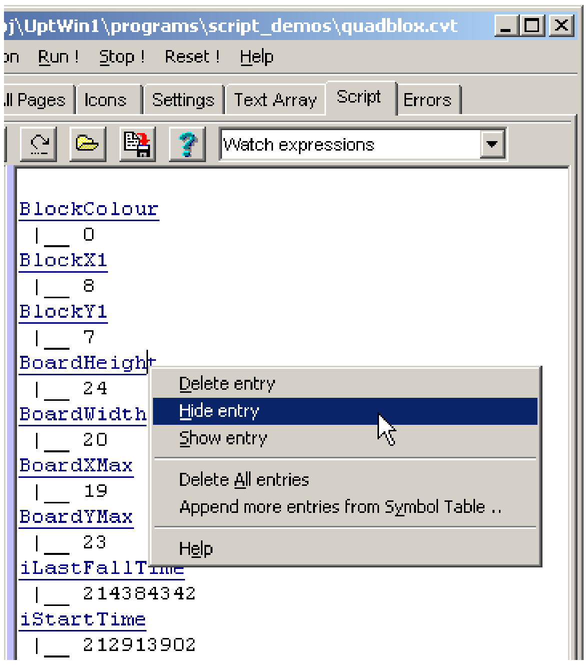
Clicking on one of the 'Values' in the symbol table opens a small popup menu as shown in the above screenshot.
'Show value in watch list' will add the symbol (usually a global script variable) to the 'watch list' shown in the next chapter.

Hint:

For devices with Ethernet connector and embedded web server (like MKT-View III), *global script variables* can also be inspected via HTML browser.

8 2.3.5 Watch List (shows values of a selection of variables)

In contrast to the *symbol table* display, the *watch list* only shows a user-defined selection of global script variables. This works with 'simple' script variables, arrays and user defined types.
(for experts: the evaluation of arbitrary expressions is not supported yet)
New items can be added to the *watch* table is via the *symbol* table, as explained in the previous chapter, or via the script editor's context menu.



```
BlockColour
  |__ 0
BlockX1
  |__ 8
BlockY1
  |__ 7
BoardHeight
  |__ 24
BoardWidth
  |__ 20
BoardXMax
  |__ 19
BoardYMax
  |__ 23
iLastFallTime
  |__ 214384342
iStartTime
  |__ 212913902
```

j\UptWin1\programs\script_demos\quadblox.cvt
on Run ! Stop ! Reset ! Help
ll Pages Icons Settings Text Array Script Errors
Watch expressions
Delete entry
Hide entry
Show entry
Delete All entries
Append more entries from Symbol Table ..
Help

Screenshot of the script editor's Watch List in the programming tool

Clicking on the name of a variable (or, in future, an expression) in the watch-list opens a small popup menu as shown in the above screenshot. The items in that menu are:

Delete entry
: deletes the previously clicked item from the watch list

Hide entry
: temporatily hides the display of the *value*, without deleting the item from the list. For arrays and larger structures, this can save a lot of screen space in the watch display.

Show entry
: Makes the *value* visible again, after it was hidden as explained above.

Delete all entries
: Quickly removes all entries in the watch list. Usefull after switching from one project to another, before adding new items to the watch list.

Append more entries from symbol table
: Switches to the symbol table, from which you can select more items (usually global script variable) to be displayed in the watch list.

Back to the overview about 'Debugging'

3. Language Reference

The scripting language was once a subset of the BASIC programming language (without line numbers), and was later modified to be more 'PASCAL'-like. Some elements were borrowed from other programming languages. From PASCAL, BASIC, and IEC 61131 "Structured Text", this language inherited the case-***in***sensitivity, so it's up to you to write keywords in upper or lower case (but please don't mix upper and lower case for keywords, and don't use "Camel Casing" if it makes no sense..). If there are CamelCased symbols in your program, they should be self-defined variables, data types, self-defined functions or procedures but *not standard language keywords* . Top-level keywords of the script language are (just for example, but also as a quick reference):

if..then..else..endif for..to..next while..endwhile repeat..until
select..case..else..endselect

proc..endproc func..endfunc

const..endconst var..endvar typedef..endtypedef

addr append float int local ptr string

int string CAN cop.(CANopen) display inet system time trace tscreen file
wait_ms wait_resume

More keywords can be found in the alphabetically sorted list.

For compatibility with the original BASIC-like language, colons (:) can be used to separate commands in one line. But we recommend to use only *one* command (function call, variable assignment, loop statement, etc) per line. Using one line per command also simplifies debugging, because you can set breakpoints only at the begin of a line.

To mimick more 'modern' programming languages, a semicolon can also be used to separate two commands in one line. But unlike "C", Pascal, and Java, the end of a line has a syntactic meaning (it also separates two commands or statements), so in *most* cases, neither the colon nor the semicolon should be necessary if you follow the style recommended above ... use ONE LINE PER STATEMENT . A few examples with the recommended style follow below.

Leading spaces have no syntactic meaning for the compiler, but you should *generously* use leading spaces (indentations) to increase the readability of your code. For example, with a bit of imagination it's obvious what this code does :

```
Sum! := 0.0 // calculate PI ...
for Loop:=1 to 10000  // do 10000 iterations
   if (Loop & 1) <> 0 // odd or even loop count ?
      then Sum! := Sum! + 4.0 / (2 * Loop + 1) // odd
      else Sum! := Sum! - 4.0 / (2 * Loop + 1) // even
   endif;
next;
print( "PI is roughly ", Sum );
```

Suggestions about the **coding style** (not mandatory, but the *highly recommended* by the author):

- Use at least two space characters per indentation level.
  Any function body (between 'proc' and 'endproc') shall be indented, too.
  Only the 'main code' (typically at the begin of the script, executed immediately after program start), and the keyword pairs **const**/endconst, **var**/endvar, **proc**/endproc, **func**/endfunc shall not be indented (because they always sit at the script's 'main level' - there are no nested functions as in Pascal).
- It's not necessary to write keywords in upper case. Keywords were sometimes written in upper case in this document, when it seemed important to mark them as such (because bold or italic characters don't work in a plain text file). Since the script editor can automatically highlight keywords, the author of the script language uses keywords in lower case, and only user-defined CONSTANTS in UPPER CASE . This is what most "C" programmers prefer.
- Don't ever use tabs in sourcecode, because different editors use different default settings (some editors use 8 characters per tab, some use 4, others 3 by default, etc...), so using tabs in sourcecodes will sooner or later turn everything into a mess, which can often be seen in open-source projects. Use two or three spaces per nesting level, and align the 'ending' statement (like **next**, **until**, **endif**) to the same column as the matching 'beginning' statement (like **for**, **repeat**, **if**) as in the example shown above.
  If you consider comments and indentation (leading spaces to emphasize nesting) a useless waste of time, stop developing software.

The following subchapters explain most of the script language's syntax elements. Special commands, keywords, and runtime library functions are explained later.

See also: Keyword list, Operators (numeric), User-defined Functions and Procedures, Program Flow Control, Other Functions and Commands .

1 3.1 Numbers

Numbers are integer by default. Their notation is usually decimal, but *hexadecimal* and *binary* is also possible (see examples below). Numbers may be integer or floating point:

- 1234 is an integer number in decimal notation (which is the default)
- 1234.0 is a floating point number (because the compiler recognizes the decimal point)
- 0xABCDEF is an integer number in hexadecimal notation (thanks to the "0x" prefix)
- 0b10000001 is an integer number in binary notation (the prefix "0b" means "binary").

Use integer numbers wherever possible. But, if an expression uses some floating point variables as input, you should also use floating point numbers (constants) because the compiler will emit floating point constants as such if it's obviously a floating point notation. This eliminates type conversions *at runtime*, and makes the script run faster. Example (with Sum! being a floating point variable, and Loop% an integer):

```
Sum! := Sum! + 4.0 / (2 * Loop% + 1)
```

does not calculate the same result as

```
Sum! := Sum! + 4 / (2 * Loop% + 1)
```

Look at the right term in the above formula: It contains only integers. When the compiler produces bytecode for the right term, it will use integer numbers because they are much faster on the target system. This also includes the DIVIDE instruction : If both operands are integers, the division will be an integer, too. If one, or both, inputs for the DIVIDE operation are floating point numbers, the division itself will be performed using a (slow) floating point operation. If you definitely need a floating point operation, use floating point numbers (constants) as in the upper example shown above. BTW, the example is taken from the application 'ScriptTest2.cvt', contained in the installation archive, which calculates the number 'PI' using the Gregory-Leibniz formula.

To convert 'binary data' (like received CAN messages) from a sequence of bytes into floating point numbers, use functions like BytesToFloat, BinaryToFloat, or BytesToDouble.

2 3.2 Strings

String constants must be enclosed in double quote characters, as in most programming languages (except Pascal).
To declare a variable as a string, use the keyword ***string***, or (if you prefer not to declare variables as in ancient BASIC dialects), use the 'dollar suffix' ($) to let the compiler know that your variable is a string.
In most places where the compiler expects strings, you may also use a *string expression* like A$+" some text "+B$.

Example (using a properly declared string variable)

```
var
   string MyString;
endvar;
...
 MyString := "This is another string";
```

The script language contains a few string processing functions, like itoa ("integer to ASCII"), hex (integer to hexadecimal ASCII), chr (turns an ASCII value into a single-character string) .

2.1 3.2.1 Strings with different character encodings

For *simple* string variables (not strings in arrays or structs), the character encoding of a string may vary, depending on the assigned value.
For simple string variables, the 'string' data type contains internal flags which specify the encoding. For example, if a string was read from a Unicode text file (using the function file.read_line), the string will contain a sequence of UTF-8 encoded characters. This way, the character encoding type is passed along with the string when calling subroutines and functions, or when assigning the string to another variable. If necessary, a string's character encoding can be queried as in the following *example* :

```
select( char_encoding( MyString ) )
   case ceDOS : // string contains "DOS"-characters (codepage 850)
       ...

   case ceANSI : // string contains "ANSI"-characters (Windows-
1252)
       ...

   case ceUnicode : // string contains "Unicode"-characters
```

```
(encoded as UTF-8)
       ...

   case ceUnknown : // the string's character-encoding is unknown
      // This means the encoding type has not been specified,
      // or doesn't matter because all characters in the string
      // have code values below 128
      // (in that case "DOS", "ANSI", and Unicode are almost
identical)
        ...

endselect;
```

Note: Just because the script language supports Unicode (to be precise, UTF-8 encoded strings) doesn't mean your application will be able to render those characters on the display ! The fonts used by MKT's LCD driver date back from the days of DOS, and only contain glyphs for the 255 characters defined in the old 'DOS' character set (Codepage 850) !
When showing an UTF-8 encoded string on the display, the firmware will try to find a match for the Unicode 'code point'. Thus, at least the common western characters (like German 'Umlauts') will appear correctly on the display, regardless of the string's **c**haracter **e**ncoding type (ceDOS, ceANSI, ceUnicode).

Since V 0.2.6 (2014-03-18), arrays and struct members of type 'string' are *always* stored as UTF-8 internally, because there are no individial character-encoding flags stored in memory for each array element.
If, for example, a 'DOS'- or ANSI-encoded string is assigned to an array element, all characters with codes > 127 will be converted to UTF-8 sequences automatically. Thus, when reading those strings from the array, they have the encoding type **ceUnicode** !

If necessary, the character encoding of *string literals* (i.e. string *constants* in the script sourcecode) can be specified by the following single-lowercase-letter prefixes. When not speficied, the compiler may decide to use ANSI, or (more likely) UTF-8:

- **a**"Text"
  "ANSI"-encoded characters (precisely: Windows CP-1252, 8 bits per character)
- **d**"Text"
  "DOS"-encoded characters (precisely: DOS Codepage 850, 8 bits per character)
- **u**"Unicode-Test"
  Unicode (precisely: characters encoded in UTF-8, with a variable number of bytes per character)

Note: The double-quote character, which marks the begin of a string constant (literal), must follow *immediately* after the prefix character (a,d,u) !

Example (assigns a Unicode string literal to a string variable) :

```
   MyString := u"Falsches Üben von Xylophonmusik quält jeden
größeren Zwerg.";
```

Wherever possible, special characters (as in the german pangram above) will be translated into their proper encoding by the compiler.

The character encoding of the *script sourcecode* is assumed to be "ANSI" (Windows CP-1252), not "DOS"-encoded characters !
This may change in the far future, if the script editor in the programming tool can be convinced to emit its content in UTF-8 instead of Windows CP-1252.
Until then, use Unicode escapes (after backslash-u) for all characters which you don't find on your PC's keyboard. Example:

```
   MyString := u"Falsches \u00DCben von Xylophonmusik qu\u00E4lt
jeden gr\u00F6\u00DFeren Zwerg.";
```

More examples can be found in the 'String Test' application.

2.2 3.2.2 String usage and storage format

Most characters in a string are internally stored as an 8-bit number. A zero-byte marks the end of a string (but you don't need to care about this, because the compiler adds the zero automatically when encountering a string constant in the sourcecode). Depeding on the string's character-encoding type, characters with a code value above 127 (!) may occupy one or more bytes in memory. 8-bit "DOS" or "ANSI" characters are stored as such in memory. A few of those 255 possible codes are reserved for special control characters, line "carriage return" and "new line" - see 'backslash sequences' further below. In addition, since September 2011, strings can optionally stored as UTF-8 sequences in memory.

During runtime, string memory is dynamically allocated from a common memory pool. The amount of memory required depends on the number of strings used in your application, and their individual lengths. For example, consider an array of structures, declared as :

```
typedef tStringTableEntry = // user defined data type..
   struct // structure for an entry in a "string table"
      int valid;    // 0: invalid or "deleted" entry, 1:valid
      int iRefNo;   // string reference number (integer)
      string sInfo; // the string itself (any length!)
   endstruct; // end of a structure definition
endtypedef; // end of type definitions


var // declare GLOBAL variables, here: an array of structs
   tStringTableEntry StringTable[1000];
endvar; // end of variable declarations
```

Initially, each `tStringTableEntry` only occupies 12 bytes (2 * 4 bytes for the integers, plus 4 bytes for a pointer to a string object in some other memory area).
Later, when the sInfo entries in the '`StringTable`' array are filled (and the strings are not "empty" anymore), each string will require *additional memory* .
In other words, the amount of memory used by your script *may increase at runtime*. Thus, to make sure your application doesn't run out of memory later, consider how many strings may be used by your application at runtime *at worst case*, and how long each of those strings may grow (because the compiler doesn't know this). Let your application run in the programming tool's simulator, and

test every function in your script. When finished, examine the peak memory usage in the debugger, and make sure the 'data memory' usage isn't critically close to the maximum.

2.3 3.2.3 String constants with special characters

The compiler doesn't know for what a string will be used later. It doesn't know anything about languages, fonts, character sets. For this reason, it doesn't try to convert any special characters (especially not German umlauts, etc). If you know the string will be displayed on the LCD screen later, using one of the DOS-compatible fonts, replace the Umlaut (etc) with the hexadecimal equivalent (backslash x followed by two hex digits, in the double-quoted string constant), or use the prefix 'd' ("DOS") before the double-quoted string to let the compiler convert the string from the sourcecode format (which is usually 'ANSI') into DOS.

Examples:

Test$:= "**`\x99`**rtliche Bet**`\x84`**ubung **`\x9A`**belkeit"; // string with hex codes for Ö, ä, Ü if a 'DOS font' is used for rendering
Test$:= d"Örtliche Betäubung kann Übelkeit hervorrufen"; // string converted into 'DOS characters' by the compiler

Hexadecimal codes for certain 'special' characters in DOS fonts (as used in most of MKT's programmable displays) :

Hex. Code	Character (here: ANSI)	Name
84	ä	a diaeresis
94	ö	o diaeresis
81	ü	u diaeresis
8E	Ä	A diaeresis
99	Ö	O diaeresis
9A	Ü	U diaeresis
DF	ß	German sharp s
.		
.		

Beware: The script language isn't aware of the font used to render a character on the screen. The compiler doesn't know what 'will happen' with a string later (if you will print it on the screen later, write it into a file, etc). Thus, **`\x94`** may print a German 'ö' (o diaeresis, or "o Umlaut") on the screen, but only if the font used to render the string is the old-fashioned 'DOS font', aka 'codepage 437' or 'codepage 850'.

A complete 'DOS' character table ("Codepage 437") can be found here . Rows and columns use hexadecimal numbers, making it very easy to find the 8-bit hex code for any desired 'special' character. The Text Screen example uses some of those characters to draw lines and boxes on the text screen.

2.4 3.2.4 Strings with backslash sequences

Besides the sequence **\x** to insert a special character (by its hexadecimal code), the following ***backslash sequences*** have a special meaning in the script sourcecode:

\\

Inserts a *single* backslash in the string .

\r

Inserts a carriage return character (aka CR, chr(13)) .

\n

Inserts a new line character (aka "linefeed", chr(10)) .

\x

Inserts an 8-bit character by its hexadecimal code (not Unicode!).
See details in chapter 'string constants with special characters' .

\u

Inserts a Unicode "character" (code point), specified as ***4-digit hexadecimal*** value.
The compiler (!) replaces the unicode value with an UTF-8 sequence.
Note that only very few of those 1114112 possible Unicode code points can later be rendered on the display !
Details in chapter 'strings with different character encodings' .

\"

Inserts a double quotation mark in the string (without a backslash, the double quote is the string delimiter, so it cannot be a part of the string itself) .

Don't confuse the backslash sequences inside the script language (listed above) with the backslash sequences in the display interpreter ! The simple control characters (like 'new line', etc) have the same meaning in both types, but the internal functions are entirely different !

See also: Invoking script functions from a backslash sequence on a display page

2.5 3.2.5 String processing

Strings can be concatenated with the '+' operator (formal "addition"). Example:

```
var
  string Info; // Deklaration einer String-Variablen
endvar;
Info := "First part";
Info := Info + " second part";
```

The following string processing functions had been implemented at the time of this writing (2013-11-05) :

`append( <destination>, <source> [, <index_variable>] )`

Appends a string ('source') to the end of another string ('destination'), or to an array of bytes.

Example 1: Appending a string to another string

```
var
  string s1,s2; // declare two string variables
endvar;
```

```
        s1  := "Don't mix apples";
        s2  := " and oranges";
        append(s1,s2);  // Append s2 to s1, result in s1 : "Don't mix
apples and oranges"
        print( s1 );    // show result on a text panel
```

In the example above (with destination = string), the third function argument ('index') is neither required nor recommended.

Example 2: Appending multiple strings to a 'binary block' (array of bytes)

```
        var
          byte TxBuffer[1024]; // declare an array of bytes
          int  TxByteIndex;    // index variable for 'TxBuffer'
        endvar;

        TxByteIndex := 0;  // begin filling TxBuffer[0] here
        append( TxBuffer, "First string.\r\n", TxByteIndex );
        // Note: append() will increment TxByteIndex by the
        //       NUMBER OF BYTES appended to the buffer !
        TxBuffer[TxByteIndex++] := 0x00;   // append a ZERO BYTE as string-
end-marker
        append( TxBuffer, "Second string.\r\n", TxByteIndex );
        TxBuffer[TxByteIndex++] := 0x00;   // append another ZERO BYTE
        append( TxBuffer, "Third string.\r\n", TxByteIndex );
        StartSendingBlock( TxBuffer, TxByteIndex/*nBytes*/ ); // user-
defined procedure
```

In this example, the 3rd function argument ('index_variable') is an integer variable which is incremented by the number of bytes appended to the destination in each call of the 'append' command. The array 'TxBuffer' is filled with multiple strings, which are delimited by a ZERO byte (as in the "C" programming language).
Because in the script language, a zero-byte also marks the end of a string, the trailing zero cannot be part of the 'netto' contents of the string itself. Thus, in the example shown above, the string delimiter is appended to the binary data block with the command

```
  TxBuffer[TxByteIndex++] := 0x00;
```

The post-increment-operator '++' increments 'TxByteIndex' by one *after* the access.
Concatenating strings via append() is faster than 'adding' them (i.e. use **append(s1,"Hello")** instead of **s1 := s1+"Hello"**), because in many cases append() doesn't need to free and re-allocate a block of memory for the string (due to the internal memory management, which allocates strings in chunks of N times 64 bytes, leaving a reserve of up to 63 characters in memory).

chr(N)

Converts an integer code (N, 0..255, usually from the "DOS" character set) into a single-character string.
If 'N' is above 255, it is assumed to be a UNICODE value, but you should better use unicode_chr(N) for that purpose.
Example: chr(32) returns the 'space' character.

`unicode_chr( N )`

Almost the same as chr(N), but unicode_chr(N) converts an integer code (N, 0..0x10FFFF) into a single-character unicode string, **regardless of whether this character can be rendered on the screen or not** ! Example: unicode_chr(0x20AC) returns a string with the internal representation (which is UTF-8) of the 'Euro Sign' character.

`CharAt( string s, int char_index )`

Returns *the code of* the N-th character in the string s. As usual, the index starts counting at *zero* for the *first* character. The result is a **32-bit Unicode value**, which, for 'normal' western characters, is the same as the ASCII value (codes 1 to 127). The 'CharAt' function is aware of the string's character encoding type, and supports UTF-8. Note that for UTF-8, there is a big difference between the 'character index' and the 'byte index'. CharAt always treats the 2nd parameter as a character index, not a byte index. If <char_index> is negative, or exceeds the length of the string, CharAt returns zero which means "there is no character at this index".

`char_encoding( string s )`

Returns the string's character encoding type. The result will be one of the following constants: `ceDOS` ("DOS"-characters), `ceANSI` ("ANSI"-characters), `ceUnicode`, or `ceUnknown` .

`ftoa( float value, int nDigitsBeforeDot, int nDigitsAfterDot )`

"**f**loating-point **to a**scii" .
Converts a floating point value (1st argument) into a **decimal string**, using the specified number of digits 'before' and 'after' the decimal dot. Example:

```
   Info := "Tire Pressure="+ftoa(fltTirePressure, 4, 1)+"
bar";
```

If the 'number of digits before the decimal dot' is larger than required, the string returned by ftoa() will be padded with *leading spaces* (not zeroes). If the 'number of digits after the decimal dot' is larger than required, the string returned by ftoa() will be padded with *trailing zeroes* (not spaces). If the 'number of digits after the decimal dot' is zero, the decimal point ('.') will also be omitted.

`itoa( int value [, number of digits])`

"**i**nteger **to a**scii" .
Converts an integer value (1st argument) into a **decimal string**, using the specified number of digits (2nd argument). Example:

```
   Info := "Timestamp="+itoa(system.timestamp,8);
```

If the value doesn't fit into the specified number of digits, the result (string) will only contain the *least significant digits*. Example:

`itoa(1234,2)` returns 24 (as string), not "1234" !

If the number of digits is not specified, or zero, itoa will produce just as many digits as required (without leading zeroes).
See also: ftoa, which emits leading spaces instead of zeroes.

`atoi( string value [, int number_of_digits [, int start_index]] )`

"**a**scii **to i**nteger" .
Converts a **decimal string** into an integer value (a numeric value), using the specified number

of digits (2nd argument, optional), or the entire string.
An optional third argument can be used to specify the start index (=index of the first character to be parsed).
Examples:

`atoi("1234567");` returns 1234567 as an integer value.
`atoi("1234567",3);` returns 123 (integer, 3 digits).
`atoi("1234567",3,2);` returns 345 (3 digits, beginning at start index 2).

Indices start counting at ZERO, not ONE. The first character in a string is at index zero.
The atoi function recognizes a trailing 'minus' character for negative numbers.

`hex( int value, int number_of_digits )`

Converts an integer value (1st argument) into a **hexadecimal** string, using the specified number of digits (2nd argument)

`strlen( string s )`

Returns the number of *characters* in the string (not 'the number of bytes', especially not for UTF-8).
Example:
`strlen("How long is this string ?")` returns 25 .

`strpos( string haystack, string needle[, int startindex] )`

Searches for the first occurrence of a search string (needle) in a larger string (haystack), beginning at the optionally defined start-index (zero-based character index).
Returns a character index (index zero = first character in 'haystack'), or a negative integer if the needle couldn't be found in the haystack.
If the 3rd argument (start index) is omitted, the function starts searching at index zero, i.e. at the first character in the 'haystack'. Example:

```
 haystack := "This test string is 38 characters long";
 needle   := "is"; // string to be found in the haystack
 i := strpos(haystack,needle);      // find the first needle
(result: i=2)
 i := strpos(haystack,needle,i+1); // find the next needle
(result: i=17)
```

`substr( string s, int start [, int length] )`

Returns a sub-string of the first argument, beginning at the zero-based character index 'start', and consisting of 'length' characters.
If the parameter 'length' is omitted, the returned string (result) includes all characters from 'start' up to (and including) the end of the source string.
If 'length' is specified and *positive*, the result will never have more than 'length' characters.
If 'start' or 'length' are *negative*, the result will be an empty string.

More examples for the string processing functions listed above can be found in the 'String-Test' application .

< To Be Completed >

See also : Keyword list , file I/O, table of contents .

3 3.3 Constants

3.1 3.3.1 Built-in constants

A few constants are hard-coded inside the script compiler. Don't rely on the actual value (that's why we don't show them in the table below).

For the sake of readability, most constants are prefixed with a lower-case 'c' (for constant). Despite that, the script compiler is case-insensitive !

Constant name	Description
.	
ceDOS	Character encoding type for strings with 'DOS' characters, actually DOS 'Codepage 850' . For historic reasons, this is the encoding used by most of MKT's build-in bitmap fonts.
ceANSI	Character encoding type for 'ANSI' characters, in fact Windows 'CP-1252' .
ceUnicode	Character encoding type for Unicode strings.
ceUnknown	Dummy character encoding type for strings which do not contain any obvious 'special characters'.
clBlack	black colour (don't care about the actual value, it may be hardware dependent) . Like the other colour constants below, this colour can be used in the setcolor-command.
clWhite	bright white . This is the second, and last, standard colour available on ALL targets.
clBlue	pure, saturated blue
clGreen	pure, saturated green
clRed	pure, saturated red
clLtBlue	Light Blue
clLtGreen	Light Green
clLtRed	Light Red
clCyan	cyan colour (mixture of blue and green)
clMagenta	red-purple, aka "fuchsia", sometimes called "pink" (which in fact it's not)
clYellow	yellow colour
clOrange	Orange
clBrown	Brown
clTransparent	Dummy colour value, only usable as back- or foreground colour of certain display elements.

csOff	Cursor Shape "Off" (text cursor invisible)
csUnderscore	Cursor Shape "Underscore" (text cursor visible, displayed as an underscore)
csSolidBlock	Cursor Shape "Solid Block" (text cursor visible, displayed as a filled block)
csBlinking	Cursor Style "Blinking" (slowly flashing text cursor)
.	
cCanRTR	RTR-flag (Remote Transmission Request) for the CAN bus
.	
cFirmwareCompDate	firmware compilation date as a string, for example "Aug 25 2010" .
.	
cPI	number "PI" (approximately 3.14159265358979323846264 3)
.	
cTimestampFrequency	frequency of the system's timestamp generator in Hertz ("ticks per second")
.	
dtUnknown	data type code for 'unknown data type'. See notes on typeof()
dtFloat	data type code for 'floating point'
dtInteger	data type code for 'integer'
dtString	data type code for 'string'
dtByte	data type code for a single 'byte' (8 bit unsigned)
dtWord	data type code for a 16-bit unsigned 'word'
dtDWord	data type code for 'unsigned 32 bit' aka 'doubleword'. Frequently used when accessing objects in the CANopen-OD via SDO.
dtColor	data type code for a colour (hardware dependent)
dtChar	data type code for a 'single character'
dtError	data type code for an 'error' (used as return value by certain functions, e.g. cop.sdo)
.	
TRUE	boolean 'true', actually 1 (one) as integer value
FALSE	boolean 'false', actually 0 (zero) as integer value
.	
keyEnter, keyEscape,..	keyboard codes. Returned by the getkey function, and used in some low-level event handlers.
.	
O_RDONLY, ...	file-open-flags. Used in the function file.open .

.	
wmXYZ	'widget messages' or, sometimes, 'windows message'. Used in event handlers. Allows the script to intercept touchscreen-, and similar low-level system events.
.	

3.2 3.3.2 User-defined constants

In addition to the fixed constants shown above, you can define your own constants.
This sometimes improves readability, especially when you need the constant's value more than once in your script.
The keyword 'const' begins a list of constant definitions, the keyword 'endconst' ends such a list.

Each constant is defined using the following syntax :
 <***constant_name***> = <***value***> ;
or (with the definition of the data type, which may be necessary if the value isn't easily recognizeable, or ambiguous) :
 <***data_type***> <constant_name> = <value> ;
or (to define a constant array, as described further below) :
 <data_type> <constant_name> [***array_size***] = <value> ;

Example (please note the coding style - indentation between const & endconst) :

```
const // define a few constants...
   C_HISTORY_BUF_SIZE = 1000; // a decimal integer
   C_CANID_RX_A    = 0x333;    // a hexadecimal integer
   C_CANID_RX_B    = 0x334;
   C_CANID_TX_ACK = 0x120;
endconst; // end of 'constant' definitions
```

User-defined constants must be defined at the begin of the script (or, at least, *before* they are used).

Notes:

- Use constants instead of 'magic numeric values' which no-one else (besides you) will understand,
  especially in long select..case statements, and as control identifiers in message handlers.
- Like the names of variables, data types, and similar elements, the name of any constant is limited to 20 characters.
- A script may use a maximum of 256 different const...endconst blocks. The number of constants (in these blocks) is only limited by the available bytecode memory, which is target specific (usually 64 kByte total *bytecode* memory).
- The old UPT *display interpreter* can access user-defined script constants without the need to prefix the constant's symbol by "script." .
  But this only works if the symbol does not exceed the maximum length of a *display variable*

(!), which is usually 8 to 16 characters.
The maximum length of a constant's name *in the script itself* is virtually unlimited.

3.3.3 'Calculated' constants (constants 'calculated' at *compile-time*, not at *runtime*)

To force the evaluation of simple expressions as numeric constants *at compile-time*, use the hash character before the constant expression in parentheses.

Example:

The expression in the command

```
N := #(1+2+3)
```

will be evaluated (calculated) at *compile-time* (by the compiler itself),
In contrast to that,

```
N := 1+2+3
```

will be calculated *at runtime*, resulting in a slightly reduced execution speed.

The compiler's own formula-evaluator is reduced to very basic calculations; it doesn't support parentheses, it doesn't support different operator precedences; and it only works with *integer* constants (*symbolic* or *numeric*).

3.3 3.3.4 Constant tables (arrays)

For some applications, it was necessary to store constants in arrays. One could use an array variable for this purpose, and fill the array at runtime using a long list of assignments. But there is a more elegant method to achive this: Constants can be defined like a formal array (and thus be accessed like an array with "read-only" access at runtime). Such constant-arrays could then be used to initialize (fill) variables-arrays in a loop, etc; because each element of the array can be accessed through an index.

Example (actually taken from the 'quadblocks' demo) :

```
CONST
  int BlockColors[7] = // seven block colours : BlockColors[0...6]
!
    { clBlue, clRed, clCyan, clYellow, clGreen, clMagenta, clWhite
};
ENDCONST;
```

See also : constants (overview), variable arrays,

4 3.4 Built-in and user-defined data types

The script language supports the following three 'basic' built-in data types :

- **`int`** (alias "integer") : signed 32-bit integer .
  This is the preferred data type for most numeric operations, and also to identify files (handles) and internet communication endpoints (sockets).
- **`float`** : 32-bit 'single precision' floating point. Contains an 8-bit exponent, a 23-bit mantissa, and a sign bit.
  This type can store fractional decimals like 0.12345. In numeric operations, it is much slower than integer due to the lack of a floating point unit in the target system.

If necessary, a floating point value can be 'constructed' from single bytes (with exponent and mantissa) via BytesToFloat() or BinaryToFloat(). Floating point numbers can be formatted into strings via ftoa() ('float to ASCII').

- **string** : a string of characters. The characters of a string are stored in a separate memory region, the length may vary during runtime.
  (In struct- and array size calculations, a string only seems to occupy four bytes in memory, but this is just a *pointer* to the actual characters.
  Details about the string type are here).

In addition to the three above 'basic' types, the following 'simple' types can be used in struct- or array declarations. When used in calculations (formulas, expressions), they are automatically converted to integer:

- **byte** : unsigned 8-bit integer (value range 0 to 255) .
  This type occupies one byte in arrays or structures. It's actually the smallest type which can be used for arrays.
- **word** : unsigned 16-bit integer (value range 0 to 65535) .
  Occupies two bytes when used in arrays or structures.
- **dword** : unsigned 32-bit integer (cannot be converted into 32-bit integer without 'losses' - only used for storage !) .
  Occupies four bytes when used in arrays or structures.

Since November 2012, type conversions between the 'basic' and 'simple' types listed above are performed automatically during *runtime* as necessary. Example:

```
var
  int   i;  // declaration of an integer variable
  float f;  // declaration of a floating-point variable
endvar;

i := 1234;    // integer value assigned to an integer variable
f := i;       // integer value automatically converted to float
// (in older versions, an explicit conversion was required here, like
//  f := float(i);   // convert integer to float, then assign to 'f' )
```

Furthermore, there are a few built-in structure definitions (belonging to the built-in data types), like :

- **tScreenCell** : Data type describing one cell of the text screen buffer . Components of this structure are:
  `.bChar` : character code, usually ASCII or even DOS character set, 0..255
  `.bFlags` : reserved for future use (2010-09-27)
  `.bFontNr` : reserved for future use (2010-09-27)
  `.bZoom` : reserved for future use (2010-09-27)
  `.fg_color` : foreground colour (each character cell has an *individual* colour !)
  `.bg_color` : background colour
- **tMessage** : Data type used by the message handling functions. Components are:
  `.receiver` : identifies the receiver of the message (if any, may be zero for 'broadcast' messages)

`.sender` : identifies the sender of the message. If the sender is not a windowed control (but the system), this value may be is zero.
`.msg` : message type code (integer). Should be one of the 'wm' ("windows message") constants, or user defined .
`.param1` : first message parameter. Usage depends on the message type.
`.param2` : second message parameter. Usage depends on the message type.
`.param3` : third message parameter. Usage depends on the message type.
For details about tMessage, see the chapter on message handling .

- **`tCANmsg`** : Data type for a single CAN message . Not to be confused with the 'tMessage' type !
  Used (as 'pointer to tCANmsg') as function argument to pass a received CAN message to a self-defined CAN-receive handler, and (optionally) as function argument for the can_transmit commmand.
  Components of the *data type* **tCANmsg** are:
  `.id` : Combination of bus-number (in bits 31+30), Standard/Extended-Flag (in bit 29), and the 11- or 29-bit CAN-ID (in bits 10..0 or 28..0).
  `.tim` : Timestamp (for *received* CAN messages, this field contains a precise timestamp filled out by the CAN driver)
  `.len` : Length of the 'netto' data field in bytes. For CAN messages, only 0 (zero!) to 8 bytes are possible.
  `.b[0] .. b[7]` : Data field as byte array (eight times 8 bits)
  `.w[0] .. w[3]` : Data field as word array (four times 16 bits)
  `.dw[0] .. dw[1]` : Data field as doubleword array (two times 32 bits)
  `.bitfield[` <Bit-Index des LSBs> **,** <Anzahl Datenbits>`]` : Bitfeld wie in den Variablen 'can_rx_msg' und 'can_tx_msg'

  To simplify communicating via J1939 protocol, the following aliases for parts of the 29-bit CAN ID were added:
  `.PRIO`: 'Message Priority'. Alias for CAN-ID Bits 26 bis 28.
  `.EDP`: 'Extended Data Page'. Alias for CAN-ID Bit 25.
  `.DP` : 'Data Page'. Alias for CAN-ID Bit 24.
  `.PF` : 'PDU Format'. Alias for CAN-ID Bits 16 bis 23.
  `.PS` : 'PDU Specific'. Alias for CAN-ID Bits 8 bis 15.
  `.SA` : 'Source Address'. Alias for CAN-ID Bits 0 bis 7.
  `.PGN`: 'Parameter Group Number', with up to 18 bits. Alias for 'DP'+'PF'+'PS'.

- **`tTimer`** : Data type for a programmable timer, with the following components:
  `.period_ticks` : Cycle duration of the timer, measured in 'Ticks' of the timestamp generator
  `.expired` : >=1 (TRUE) if the timer is expired, otherwise 0 (FALSE)
  `.ts_next` : Current value of the timestamp generator (system.timestamp) at the *next planned expiration* of this timer
  `.running` : >=1 (TRUE) if this timer is currently 'running', otherwise 0 (FALSE)
  `.user` : A freely useable 32-bit integer value assigned to this timer,
  for example an event counter or an index (see timer event demo)

The **tTimer** type is used by the setTimer command, and (passed as 'pointer to tTimer' in the argument list) when periodically calling a timer event handler.

Some of the script language functions may return *different* data types as their 'return value'. If the caller needs to examine *the type of* the result, he can use the typeof operator (operating on the returned value), and use a select..case statement to implement different processing for each data type. For this purpose, use symbolic constants like dtFloat, dtInteger, dtString, etc in the case marks; and declare the variable (for the return value) as 'anytype':

- **anytype** : Placeholder for the data type to declare a variable (or function argument) which can accept 'any type'.
  After assigning a certain value to such a variable, the actual type can be examined with the typeof() operator.
  Example:

```
var
     anytype result;
endvar; // end of variable declarations
     ...
     result := cop.sdo(0x1234, 0x01);  // read something via CANopen
(Service Data Object)
     select( typeof(result) )  // what's the TYPE OF the returned value ?
        case dtInteger:  // the SDO transfer delivered an INTEGER
           ...
        case dtByte:     // the SDO transfer delivered a BYTE
           ...
        case dtString:   // the SDO transfer delivered a STRING
           ...
        case dtError:    // the SDO transfer returned an ERROR CODE
           ...
     endselect;
```

Besides the data types listed above, own data types can be defined. Example:

```
typedef
  tHistoryBufEntry = // this is the name of a user-defined data type
      struct
           int iRefNo;    // 1st member: an integer variable
           int iSender;   // 2nd member: another integer
           float fUnixTimestamp; // 3rd: a floating point variable
           string sInfo;  // 4th member: a string
      endstruct;
end_typedef; // alias endtypedef, end of the data type definitions
```

Notes:

- A block of 'typedefs' may define more than one data type; the semicolon is required to separate the entries.
- Types must be *defined* before they can be used to *declare* variables. Put all typdefs at the begin of your program.
- using a lower-case 't' as suffix for own *type definitions* is not mandatory, but recommended to make the script easier to read.

- components within a type definition must be separated with semicolon (a colon doesn't work here) .
- The keyword **typedef** begins a *type* definition, the keyword **end_typedef** (alias endtypedef) ends it.
- The keyword **struct** begins a *structure* definition, the keyword **endstruct** ends it.
- Structures can only be accessed component wise. Copying a complete struct to another as in "C" is not possible (yet?).

At the moment, structures ("structs") can only be composed of basic data types. Nested structures, and structs with arrays as components, were just future plans (at the time of this writing).

See also: var..endvar to define a list of *global* script variables.

5 3.5 Variables

The script language supports local and global variables (more on local vs global in the next chapter).
In addition, the script also has limited access to variables of the display interpreter (defined for the UPT display application).

As in most programming languages, variables *should* be declared before using them, but this is not necessarily the case (for BASIC-compatibility).

Deprecated (not recommended for new developments): For non-declared 'automatic' variables, the data type was defined by their ***suffix***, like '%' for "integer", '&' for "long integer", '$' for "string", and '!' for floating point.

Again, using 'non-declared' variables is deprecated, and should be avoided. Instead you should *declare* variables properly (with data type) before using them, as explained in the following chapters.

Hint for developers:

Global variables can be inspected at runtime by the built-in debugger (values listed in the symbol table) or remotely using the device's embedded web server / 'script' page.
Local variables cannot be inspected that way because they only exist during a function call, and may exist in multiple instances on the stack.

5.1 3.5.1 Variable declarations in the script

As mentioned in the previous chapter, any kind of variable should be *declared* along with its type, prior to using it.

3.5.1.1 **Global script variables**

Only variables with user-defined types, array, or anything else which is not a simple variable must be declared before use. This is what the keyword 'VAR' is for. After the 'VAR', place the type name *before* the name of the variable. A variable-definition-block must end with the 'ENDVAR' keyword (alias END_VAR). Example for the declaraction of some global variables (please note the coding style - indentation between var & endvar) :

```
var // global variables (accessable from all subroutines) ...
   int nHistoryEntries;  // declares an integer variable
   tHistoryBufEntry History[100]; // declares an array of tHistoryBufferEntries
    // Note: the indices for an array with 100 entries run from 0 (ZERO) to 99 !
```

```
 logged: // The following variables may be recorded by the built-in CAN logger:
    // Signals from J1939 PGN 61443 = "Electronic Engine Controller 2" :
    int   AccelPedalKickdown;   // SPN 559  "Accelerator Pedal Kickdown Switch"
    int   AccelPedalPosition1;  // SPN 91   "Accelerator Pedal Position 1"

 private: // The following variables shall NOT be 'logged':
    int   iSomeInternalStuff;
    ...
endvar;
```

Between the keywords '**var**' and '**endvar**' (i.e. within the declaration of *global* script variables), the following attributes can be speficied (they apply to the varibles declared ***after*** the attribute):

private:
The subsequent variables shall only be accessable *inside* the script;
they shall *not* appear in the selection lists for defining display pages,
and they shall *not* be logged.

public:
The subsequent variables shall be accessable *outside* the script, too.
The programming tool will include them in the selection list for defining display pages.

logged:
The subsequent variables *may*(*) be recorded by the logger which is integrated in certain devices.
To end a list of 'loggable' variables, use the attribut 'private:'.

(*) The 'logged' attribute doesn't mean subsequent variables are *always* logged.
In addition, the option + **script variables declared as 'logged'** must be set in the CAN-Logger Configuration to log such variables, besides CAN-messages and GPS data.

Global variables (regardless of being 'private' or not) can be inspected during runtime by the built-in debugger (values listed in the symbol table) or remotely using the device's embedded web server / 'script' page.

3.5.1.2 Local script variables

Inside user-defined procedures or functions, ***local*** variables can be declared (see following example). Local variables use the *stack* for storage, thus they only 'exist' inside the procedure / function until it returns to the caller. Example:

```
proc Test
   local int x,y,z; // define three local integer variables
      ....
    print( x,y,z );
endproc // local variables (x,y,z) cease to exist at this point
```

Note that there is no 'endlocal' statement, because LOCAL only applies to the declarations in the same line, right next to the LOCAL statement. To avoid running out of stack memory (which is limited to a few hundred entries), try to keep down the amount of local variables in your code.

Especially if such procedures call each other recursively, they will consume a lot of precious stack memory, because each new call occupies one 'stack frame' (which contain function arguments and local variables) on the stack.
As soon as a function returns to the caller, its local variables (in fact stack locations) are freed automatically, and their addresses become invalid.

See also: Debugging ... Stack Display

5.2 3.5.1.3 Pointers (pointer variables and address operations)

Similar as in the 'C' programming language, variables can be declared as 'Pointers' for *special purposes*. But, as in 'C' (and as explained further below), pointers must be treated carefully, because the runtime system cannot check (exactly) if a pointer points (or *still points*) to a valid location, and if the type of the pointer is really compatible with the target location.

Example for the declaration of a variable with the type 'Pointer to Integer':
  int **ptr** myPointerToInteger; // declaration of a typed pointer

Purposes of pointers may be...

- the manipulation of 'binary data blocks' (which are neither simple array nor described as a structure)
- passing large data blocks 'by reference' (to avoid lengthy and slow block-copy operations)

When using pointers, ...

- use them only if really necessary
- make sure that the pointer's address is still valid when you *de-reference* ("use") it
- beware that the address of any local variable gets invalid, as soon as the function (in which the local variable was declared) returns to the caller
- thus, only set pointers to global script variables (which remain at fixed addresses)

In many cases, pointers can be replaced by arrays or self-defined data types. Main advantage of a pointers: In the script language, a pointer only occupies four bytes (32 bits) in memory, thus it can be easily copied and passed as argument to subroutines (functions, procedures, event handlers). Depending on the size of the target object, copying a pointer can be ***much*** faster than copying (duplicating) the object itself.
For linked lists, trees, and similar data structurs, pointers are (almost) inevitable.

5.2.1 Assigning the *address* of a variable to a pointer

To set a pointer to a certain variable, ***the address*** of a variable (or whatever) must be taken. This can be achieved by the operator function **addr**:

The term addr(`<variable>`) returns *the address of* the variable inside the argument list.

Example for the declaration and initialisation of a pointer, to have it pointing to a 'simple' variable :

```
var // declare global variables...
```

```
  int myIntegerVar;   // declare an integer variable
  int ptr myPtrToInt; // declare a variable with a pointer to an
integer value
endvar;

myPtrToInt := addr( myIntegerVar ); // assign address of
'myIntegerVar' to pointer 'myPtrToInt'
```

To de-reference a pointer (i.e. "access the object to which the pointer points"), append a formal array index in squared brackets after the name of the pointer (unlike "C", there is no '*' operator for this).
The formal array index is almost always zero, which means 'use the element to which the pointer points, without offset'.
In contrast to 'real' arrays, the runtime system cannot check the validity of the pointer's formal array index (=offset), because a pointer only carries a data type along, but not an array size.
Thus, as in "C", the *developer* is responsible for the validity of a pointer !
If the formal array index is non-zero, it will be multiplied by the size of the pointer's data type, to calculate the *address offset* which is added to the address to dereference the pointer (for example, multiply the offset by four for a 'pointer to integer'). For typeless pointers ("pointers to anything"), the formal array index must only be zero.
Examples to de-reference a pointer (aka 'access the data via pointer'):

```
  myPtrToInt[0] := 12345; // pointer access as a formal array,
here: index zero = first array element
  myPtrToInt[1] := 0; // here ILLEGAL, because in this example
myPtrToInt only points to ONE integer value !
```

When accessing a single component of a structure (also a user defined type) via pointer, the pointer will automatically be dereferenced (unlike "C", where you'd use '->' to access a struct component via pointer, and '.' to access a struct component directly).
Example to access a component of a structure *via pointer*:

```
typedef // define data types and structs...
  tMyStruct = struct
    int iRefNo;
    string sName;
  endstruct;
end_typedef;

var // declare global variables...
  tMyStruct myStruct;   // declare a variable of type 'tMyStruct'
  tMyStruct ptr myPtr;  // declare a pointer to a 'tMyStruct'
endvar;

myPtr := addr( myStruct ); // take address of 'myStruct' and
assign it to pointer 'myPtr'
```

```
myPtr.iRefNo := 12345;      // actually sets myStruct.iRefNo
myPtr.sName := "Hase";      // (in 'C' this would be myPtr->sName)
```

5.2.2 Passing function arguments (parameters) via pointer

When passing arguments to functions, procedures, or event handlers, pointers are often used instead of passing larger structures directly ('pass by reference' instead of 'pass by value'). The reason is that a pointer only occupies four bytes in memory, thus a pointer can be passed to a subroutine much faster than copying an entire structure in memory. More on this in the chapter about User-defined functions and procedures.

For example, a CAN-Receive-Handler uses *a pointer to* a CAN message as argument, which actually points to the CAN message in the system's CAN-receive-FiFo, rather than copying the entire CAN message to the stack (i.e. call-by-reference, not call-by-value).

5.3 3.5.2 Accessing script variables from a display page

Using the prefix "script.", any 'simple' *variable in the script* can be read *from the display application*. This may be necessary in a few cases. Generally yhou should not access script variables directly from the display pages, to avoid inconsistencies (reason: display application and script are not synchronized *per se*, the script runs 'in the background', possibly in a multitasking environment, and the display application 'doesn't know what the script is doing' when it accesses the script's variable). Some of the script examples use this feature to inspect variables on the terminal's screen.

See also:

- Synchronisation between script and display by pausing the display (while the script 'calculates a new set of results')
- More about interaction between *script* and *display application* :
  - Accessing ***display variables*** from the script
  - Accessing ***script variables*** from the display interpreter
  - Invoking ***script procedures*** from the display interpreter
  - Invoking ***script functions*** from display pages (to retrieve a text strings for the display, used for internationalisation)

3.5.3 Accessing display variables from a script

In some cases, the script code may need to read or modify the value in one of the *display variables*. This is more complex than you may guess, because the script may be called while the display is being updated (especially if the display update is quite slow). For this reason, any access to a display variable from the script code must use the prefix "display." before the name of a display variable. The system will make sure that the value of a "display variable" cannot be modified during the display page update, or during the handling of programmed 'display events' .
Example (with 'Oeldruck' being a *display* variable connected to a 'CANdb'-Signal) :

```
  if ( ! display.Oeldruck.va ) then
     print( "Oil pressure isn't valid !");
  else if ( display.Oeldruck < 1.2345 ) then
     print( "Oil pressure is too low !");
  endif;
```

Note: For reasons explained above, accessing display variables from a script may slow down the script significantly (because it may have to "wait" for the display).
Never use display variables inside the script for anything else than 'showing them on the display' !

See also:

- Controlling the programmable display pages from the script (page switching, etc)
- More about interaction between *script* and *display application* :
    - Accessing ***display variables*** from the script
    - Accessing ***script variables*** from the display interpreter
    - Invoking ***script procedures*** from the display interpreter
    - Invoking ***script functions*** from display pages (to retrieve a text strings for the display, used for internationalisation)
- Keywords
- Examples
- Overview (of this document)

3.6 Arrays

Variables and constants can both be declared as arrays. The syntax is similar to the "C" programming language (there is no "array" keyword) :

Squared brackets (not parentheses!) are used for array sizes, and -later, when accessing the array elements- as the array index.

As already mentioned in the chapter about variable declarations, array indices run from zero to < array-size MINUS ONE > !

Example: 3D-Array, organized in "pages", "lines", and "columns"

```
VAR
  int ThreeDimArray[10][20][30];
ENDVAR
    ...
  z := 1; // "page" index, valid: 0..9
  y := 2; // "line of page", valid: 0..19
  x := 3; // "column of line", valid: 0..29
  ThreeDimArray[z][y][x] := 1234;
```

Notes on arrays:

- The maximum number of dimensions in an array is THREE.
  Four-dimensional arrays are impossible.
  Arrays of arrays are also impossible, pointers *to* arrays are impossible, and pointers *inside* arrays are impossible (at least, as of 2010-10-14).
- Certain data types (like strings) are problematic in arrays, because a 'string' is in fact just a pointer to a different memory area.
  Thus, the contents of an array cannot easily block-copied :
  Block-copying an array of strings would only copy their addresses, but would not duplicate their contents (copy characters).
- For that reason, partial array references (as in C) are forbidden.
  Trying to "copy" an entire page (1st dimension of the sample 3d-array shown above) like
  `ThreeDimArray[z] := ThreeDimArray[z+1]`
  is impossible (at least, as of 2010-10-14) .
- The content of an entire array can be inspected at runtime in the programming tool:
  Enter the name of the array (as a global script variable, without indices) in the Watch List.
- An array of BYTES can be used as a storage for any kind of 'binary' data. The append() command can be used to append strings of characters (without the trailing zero, which is specific for the script language) to such an array.
- When filling an array element-by-element, consider using the '++' operator to increment the index variable:
  ```
   TxBuffer[TxByteIndex++] := 0x00; // append a ZERO BYTE to the
  array
  ```

An example demonstrating the use of arrays can be found in the test application "ScriptTest4.cvt" (contained in the installer, subfolder 'programs').
A more sophisticated example using arrays of constants and variables is in the 'quadblocks' demo.

6 3.7 Operators

The script language uses almost the same numeric operators as the UPT display interpreter (even though the internal evaluation of those operators are totally different - see the chapter about bytecode if you are curious about the details). At the time of this writing (2013-11-08), the following operators have been implemented :

Operator	Alias	Precedence	Remarks
^	POW	5 (highest)	reserved for 'A power B' (not bitwise EXOR!)
*		4	multiply
/		4	divide
%	MOD	4	modulo (remainder)
+		3	add
-		3	subtract
<<	SHL	3(?)	bitwise shift left
>>	SHR	3(?)	bitwise shift right
==	= (*)	2	compare for 'equality'
!=	<>	2	compare for 'not equal'
< , > , ..		2	other compare operators
||	or	1 (lowest)	logical (boolean) OR
&&	and	1	logical AND
|	BIT_OR	1	bitwise OR
&	BIT_AND	1	bitwise AND (a binary operator)
EXOR		1	bitwise (!) EXCLUSIVE-OR
!	NOT	1	boolean negation
~	BIT_NOT	1	bitwise NOT (complement)
addr(variable)	& (prefix)	1	Retrieve *the address* of a variable
++ (suffix)		1	post-increment
-- (suffix)		1	post-decrement

(*) Avoid using a single '=' character as the 'compare-equal' operator. You should also avoid using the single '=' character as the assignment operator.

Suggestion to resolve this ambiguity :

Use ':=' to assign a value (right of the operator) to a variable (left of the operator). This operator is borrowed from PASCAL.

Use '==' to check for equality. This operator is borrowed from the "C" programming language (which also inspired Java many years later).

Without this, the compiler would have to guess if '=' means "assign" or "compare for equality". It usually makes a correct guess, at least in the obvious cases.

In case of doubt about operator precedence, use parentheses. Don't leave anything to fate ! Especially for the bitwise and boolean "AND" and "OR" operators, there are no different precedence levels (there is no "AND" before "OR" as in 'C' yet), so you are forced to use parenthesis in cases like this:

Warning := EngineRunning **AND** ((WaterTemp < 5) **OR** (WaterTemp > 95))

See also: Keywords , overview .

6.1 3.7.1 The 'address taking operator' ('&' or 'addr')

The ampersand, when used as unary operator, takes the address of the object right next to it. This is similar as the 'address taking operator' in the "C" programming language:
If 'MyVariable' is the name of a variable (local or global), then **&MyVariable** retrieves the address of that variable in memory.
This operator is typically used when passing arguments to functions ***by reference*** rather than ***by value*** (i.e. pass the *address of something* to a subroutine instead of passing a copy of the value itself on the stack).
For example, see inet.recv() . The 'output arguments' are in fact *addresses*, thus the name of variables must be prefixed by the address-taking operator in this special case.

We suggest to use the more descriptive 'addr()'-operator instead of the ampersand. Both 'address taking' operators have the same purpose.

6.2 3.7.2 Increment- and Decrement-Operator ('++', '--')

The '++' operator, when used on the *right* side of an integer variable inside an expression, **increments** the value of the variable *after* retrieving the current value.
This is similar as the 'post-increment operator' in the "C" programming language.
In a similar fashion, '--' **decrements** the value of the variable *after* retrieving the current value.
Example:

```
j := i++;  // first copy 'i' to 'j', then increment 'i' by one
```

The above code has a similar effect (but runs faster) as the following:

```
j := i;   // copy 'i' to 'j'
i := i+1; // increment 'i' by one
```

The '++' operator is often used to increment the index when filling arrays. The index variable is initially set to zero, and then incremented by one whenever a new item was appended to the array. One of the examples for the append() command also uses the '++' operator to append data to an array:

```
TxBuffer[TxByteIndex++] := 0x00;   // append another ZERO BYTE
```

In this example, 'TxByteIndex' is the array index for filling data into a byte array ('TxBuffer'). With each byte appended to the array, 'TxByteIndex' is incremented by one **AFTER** being referenced as index into the array.

Back to the overview of operators .

7 3.8 User-defined functions and procedures

Procedures can replace 'gosub-return'-subroutines since 2010-10 . Their main purpose is to give a script a cleaner structure, allow parameter passing (through a well-defined argument list after the procedure name), and allow recursive algorithms (by virtue of local variables on the stack). Unlike user-defined functions, procedures do not return a value directly to the caller, and thus cannot be used in expressions.

7.1 3.8.1 User-defined procedures

Here is a simple example for a user-defined, *recursive* (*) procedure which prints a decimal number to the text screen (taken from the TScreenTest example).
Please note the indentation between 'proc' and 'endproc', and use a similar coding style in your own scripts. The compiler doesn't care for these leading spaces, but they make the sourcecode much easier to read.

```
//---------------------------------------------------------
proc PrintDecimal( int i )
  // Simple RECURSIVE procedure to print a decimal number .
  if( i>10 )
    then PrintDecimal( i / 10 ); // print upper digits,
recursively
  endif;
  print( chr( 48 + (i % 10) ) ); // print least significant digit
endproc; // end PrintDecimal()
```

Example to call the procedure defined above (explained in the chapter about recursive calls) :

```
N := 123456;
PrintDecimal( N );  // call user-defined procedure
```

Internally, shortly before the call of the procedure, the value of N is read, pushed to the stack. The procedure (or function) uses the value on the stack like a local variable. Inside the procedure, 'i' (=name of the local variable, here: function argument) has its own storage location for each new call.
If you're interested in the details: The virtual machine which executes the script code uses a register called BP (base pointer) to access function arguments as well as local variables.

In contrast to the rule '*exactly one line of sourcecode per instruction*', the headline of a user-defined procedure or function (~~ 'function prototype' in "C") may extend over more than one line of sourcecode. Example (from the 'TimeTest' demo) :

```
//---------------------------------------------------------
proc SplitUnixSeconds(
        in int unix_seconds, // one input, six outputs...
        out int year, out int month, out int day,
        out int hour, out int min, out int sec )
// Procedure to split 'Unix Seconds' into
// year (1970..2038), month (1..12), day-of-month (1..31),
// hour (0..23 ! ), minute (0..59), and second (0..59) .
```

7.2 3.8.2 User-defined functions

User-defined functions work almost the same as user-defined procedures. The main difference is that a function returns a value to the caller.
Simplistic example: User-defined function to add two integer values.

```
//----------------------------------------------------------
func Sum2( int a, int b) // adds two integers, returns the sum
   return a+b;
endfunc;

Summe := Sum2( 1, 2 );   // invoke the user-defined function
```

Note that the function header doesn't specify a type for the return value ! The reason is just a *future plan*:
Script functions may return different types of results, similar to JavaScript (not Java).

In a user defined function, the 'return' command, followed by a value, will return to the caller with the specified value as the function's "result" (aka "return value").

If the program counter reaches the end of a function ("endfunc") without a 'return' instruction, the function returns 0 (zero) as an integer value.

Functions with certain 'special names' can be called as event handlers. In that case, the function call will **interrupt** the normal program flow (at any point), and the function's return value defines whether the event shall be processed by the system (using the system's default message handler) or not.

You will find other (less simplistic) user defined functions and procedures in the script examples.

7.3 3.8.3 Invoking script functions through a backslash sequence from a display page

Since 2011-08, user-defined functions (written in the script language) can be invoked from a display page, to replace the text in one of the display page's format strings. For the display interpreter, the function call must be embedded in a backslash sequence in a display format string (so the display interpreter recognizes it as a function call, not ordinary text).

Syntax (in the *format string* of a display line definition):

> **\script.**<*function_name*>**(**<*arguments*>**)**
> where <function_name> is the name of a user defined function (defined in the script language);
> and <arguments> are the arguments passed to the function. The number of arguments, and their data types, must match the called function - see example below.

Example (for the format string in a display page definition):

> **\script.GetText(123)**
> where GetText is the name of a user-defined function, written in the script language, which takes an integer argument (here: a 'text reference number') as input, and returns a string as return value. The maximum string length returned to the display-interpreter this way is 1024 characters (limited by the display's static string types, not by the script language). This

example is based on the 'multi language demo' (script_demos/MultiLanguageTest.cvt) :

```
//--------------------------------------------------------------
func GetText( int ref_nr ) // User defined function .
// Returns strings in different languages .
// Input: ref_nr = reference number for a certain text,
//          may run from zero to 9999 .
// Currently selected language in variable 'language',
//      which may be 10000(=LANG_ENG) or 20000(=LANG_GER), etc.
  select( ref_nr + language )
....
    case #(123 + LANG_ENG): return "onehundredandtwentythree";
    case #(123 + LANG_GER): return "Einhundertdreiundzwanzig";
....
    else: return "missing translation, ref_nr="+itoa(ref_nr);
  endselect;
endfunc;
```

Note: Similar as for event handlers, the function invoked from a backslash sequence should return 'as fast as possible' to the caller. Long loops, file I/O, and other slow operations must be avoided. Otherwise the device would appear to be non-responsive (or, from the operator's point of view, "reacts sluggish" or even "crashes"). A watchdog in the runtime system terminates the function call, if the function doesn't return to the caller within a few hundred milliseconds (time specified in the chapter about event handling). This also applies to functions invoked from the display interpreter via backslash sequence, etc. Such a limitation does not exist in the normal script context ("main loop"), due to the pseudo-multitasking. If the *maximum time* is not sufficient for whatever-your-function-needs-to-do, and if *nothing else helps*, you can avoid this by feeding the watchdog in the script yourself - but beware of the consequences (sluggish response to user actions, protocol timeouts, etc).

See also: More about interaction between *script* and *display application* :

- Accessing ***display variables*** from the script
- Accessing ***script variables*** from the display interpreter
- Invoking ***script procedures*** from the display interpreter

3.8.4 Invoking script procedures from the *display interpreter*

In a few rare cases, you may need to invoke a procedure (or a function) written in the *script language* from a display interpreter commandline. For example, call your script from the 'Reaction' of a programmable button:

Definition of a button (in the UPT display application):

```
    \btn($2,"German",60,script.SetLanguage(LANG_GER))
```

The prefix '**`script.`**' tells the display interpreter that a procedure (or a function) written in the script language shall be called.
In the example shown above 'SetLanguage' is the name of a user-defined procedure, called when the button is pressed.

There a some restrictions of 'what may be done' in a procedure (or function) called from the display interpreter. Most important: The procedure must not block the caller for more than a few dozen milliseconds, as explained for Event Handlers written in the script language. Reason: In contrast to the normal script execution, the display interpreter cannot 'wait for a long time' when updating a display page, or checking for display-events - the system would seem to 'freeze' from the operator's point of view. For details about the maximum time spent in the called function, see system.feed_watchdog.

A complete example can be found in the 'Multi-Language-Test'.

See also: More about interaction between *script* and *display application* :

- Accessing ***display variables*** from the script
- Accessing ***script variables*** from the display interpreter
- Invoking ***script functions*** from display pages (to retrieve a text for the display, used a backslash sequence in the display element's format string)
- Event handling in the script language (as a replacement for the 'events' defined in the UPT display pages)

3.8.5 Input- and output- arguments

By default, all parameters in a procedure's (or function's) argument list are "inputs", which means the procedure can read their value, but cannot modifiy the value in the caller's variable. Only arguments after the keyword 'out' in the formal argument list may affect the caller's variable. Arguments declared as 'in' (input), or without in / out, cannot affect the caller's variables in any way (last not least because the script language doesn't support pointers or call-by-reference yet). Example :

```
proc AddInteger( in int A, int B, out int Result )
  Result := A + B;
endproc
...
VAR
  int N;
ENDVAR;
AddInteger( 1,2, N ); // CALL of the procedure defined above. Output copied to 'N'
when returning .
```

The 'in' keyword was only added for clarity, to emphasize that an argument is NOT an 'output' but 'input' (read-only from the procedure's point of view) .

Note that the parameter passing mechanism for arguments declared with the 'out' keyword doesn't have anything to do with pointers, or 'call-by-reference' as known from other programming languages. In fact, arguments declared as outputs are 'written back' to the variable (from which they were read before the call) *by the caller* ! This has the side effect that the modified output value does NOT have an effect on the caller's variable until the procedure (or function) ***returns***. In other words, all 'outputs' become effective at the same time --- in the moment the function / procedure returns to caller !

7.4 3.8.6 Recursive calls

In this context, a *recursive call* means that a procedure (or user-defined function) may call itself ... as long as there is sufficient stack memory available. For each call instance, a new set of local variables (which includes the parameters in the argument list) is allocated on the stack, and freed when the procedure (or function) returns to the caller.

To understand recursive function calls, look at the PrintDecimal example from a previous chapter again :

```
proc PrintDecimal( int i )
  if( i>10 )
    then PrintDecimal( i / 10 );
  endif;
  print( chr( 48 + (i % 10) ) );
endproc;
```

In a sample call, PrintDecimal(123456), the first instance is created with i = 123456 (as a local variable on the stack). Because 'i' is greater than ten, the procedure calls itself (= recursion !) with i = 12345 . For the second (recursive) call, a new instance is created, occupying additional stack space. In that instance, 'i' is still greater than ten, so the recusion continues, until 'i' is less than ten (actually, it will be one then, which is the most significant digit, which is printed to the screen first). This results in the following 'call history' (-> means "calls", <- means "returns to caller") :

```
PrintDecimal( 123456 )
  -> PrintDecimal( 12345 )
      -> PrintDecimal( 1234 )
          -> PrintDecimal( 123 )
              -> PrintDecimal( 12 )
                  -> PrintDecimal( 1 )
                       (no further recursion; prints "1")
                  <- (procedure returns to the caller)
                 (caller now prints 12 modulo 10 = "2")
              <-
             (caller now prints 123 modulo 10 = "3")
          <-
         (caller now prints 1234 modulo 10 = "4")
      <-
    (caller now prints 12345 modulo 10 = "5")
  <-
 (first instance finally prints 123456 modulo 10 = "6")
<-
```

Recursive calls can also involve more than one procedure, calling each other. Even though they may be an elegant solution in some cases, their stack usage is hard to predict. So, in many cases, loops and similar constructs (see next chapter) are a better, more 'robust' alternative.

8 3.9 Program flow control

The script language supports program flow commands like

- if..then..else..endif
- for..to.. [step] .. next
- while .. endwile (checks the condition at the begin of the loop body)
- repeat .. until (checks the condition after the loop body, i.e. loop runs at least once)
- select .. case .. else .. endselect
- ~~goto (jumps to a label within the same function ... please forget about line numbers!)~~
- ~~gosub .. return (deprecated, use procedures wherever possible)~~
- stop : stops the execution of the script's "main program". Only special calls (from event handlers) are possible after this command.

Note: As in BASIC and IEC 61131 "structured text" (and in contrast to languages like "C" and Java), built-in commands and keywords are case-insensitive. Some users prefer to write keywords in all UPPER CASE. See notes about case-insensitivy and recommended coding style.

In the broader sense, user-defined functions and procedures are also suitable (*very* suitable) to control the program flow. Since the introduction of procedures and functions, you should not use 'goto', 'gosub' and 'return' in a new appication. They only remain part of the language for backward-compatibility.

8.1 3.9.1 if-then-else-endif

Simple example:

```
if A<100 then
    do_something_if_a_is_less_than_onehundred ;
     ...
else
    do_something_else ;
     ...
endif;
```

In contrast to the rule '*exactly one line of sourcecode per instruction*', the condition between **if** and **then** may extend over more than one line of sourcecode. This allows complex and nested constructs as presented in the 'examples' section of this document.

To simplify a chain of 'else','if' and 'endif', the 'elif' (else-if) command can be used as in the following example:

```
if ( A < 0 ) then
    print( "Negative !");
elif ( A==0 ) then
    print( "Zero");
elif ( A==1 ) then
    print( "One");
elif (A >= 2) and ( A <= 3 ) then
    print( "Two to Three");
elif (A == 5) or (A == 7) then
    print( "Five or Seven");
else
    print( "Some other value (",A,")" );
```

```
     endif;
```

8.2 3.9.2 for-to-(step-)next

Loop using an index variable, which runs from the specified start value to the specified end value.

Simple example:

```
for I:=1 to 100
  do_something_a_hundred_times
next;
```

Optionally, the stepwidth of the index variable (in the example, "I") can be specified, using the STEP keyword:

```
for I:=0 to 200 step 2
  do_something_a_couple_of_times
next;
```

If the counter-variable shall be decremented rather than incremented, use a negative STEP value (the compiler cannot use a negative stepwidth automatically, because he doesn't see the start- and end value at compile time). For more examples, study the 'LoopTest' application.

8.3 3.9.3 while..endwhile

Syntax:

```
while <condition>
  <statements>;
endwhile;
```

Loops while the condition, checked at the beginning of each loop, is TRUE (non-zero) .

Simple example:

```
I:=0; // make sure 'I' starts at some defined value
while I<100
  I := I+1 ;
  some_other_statements ;
endwhile; // .. or END_WHILE for IEC61131-similarity
```

Note that the statements inside a WHILE-ENDWHILE loop may be executed ZERO times.

8.4 3.9.4 repeat..until

Syntax:

```
repeat
  <statements>;
until < stop_condition >;
```

Loop with a STOP-condition, checked at the end of the loop (specified after the keyword "until").

Example:

```
I:=0; // make sure 'I' starts at some defined value
repeat
  I := I+1;
  some_statements
until I>=100;
```

Note that the statements inside a REPEAT-UNTIL loop are executed *at least once* !

8.5 3.9.5 goto

Unconditional jump. Try to avoid using 'goto' whereever possible ! Using too many goto instructions in your code will turn it into a difficult-to-maintain nightmare, aka 'Spaghetti-Code'. Or, as Niklaus Wirth put it, "GOTO Considered Harmful" . To discourage the use of 'goto', it doesn't not work *inside* user-defined functions and procedures.
Simple, and deliberately poor, example:

```
IF divisor==0
  THEN GOTO ErrorHandler;
  ELSE quot := dividend / divisor;
ENDIF
```

... some other code ***in the same subroutine*** ....

```
ErrorHandler: // we shouldn't get here...
  Info$ := "Something went wrong";
  STOP
```

8.6 3.9.6 gosub..return

Simple subroutine call without parameter passing. Deprecated, see note below (use procedures instead of 'gosub' / 'return') .
"Gosub" works a bit like "goto", but places the address of the next instruction to be executed (in the caller) on the stack.
"Return" returns to the address on the top of the stack, i.e. continues execution at the caller's next instruction.

Note: Unlike user-defined procedures, stoneage gosub-return subroutines don't have their own stack frame; therefore you cannot define local variables in such subroutines.
Wherever possible, use procedures, and avoid gosub-return (as well as you should avoid 'goto').
'Gosub' only exists for compatibility reason.

8.7 3.9.7 select..case..else..endselect

This construct may replace a long nested sequence of if-then-else statements, but only compares INTEGER CONSTANTS in the case-marks.

Since September 2013, 'case' supports an extended syntax. A complete *range* of values can be specified as "**case** <Value1> **to** <Value2>". The code after that case-label is executed if the select-value is between (and including) 'Value1' and 'Value2'. Thus, "case 4 to 6:" in the following example checks if 'X' is 4, 5, or 6.

Simple example using a bit of pseudo-code ("do_something") :

```
X := can_rx_msg.id;
select X
   case 1 :      do_something_if_X_equals_one();
   case 2 :      do_something_if_X_equals_two();
   case 3 :      do_something_if_X_equals_three();
   case 4 to 6 : do_something_if_X_is_between_four_and_six();
   else   :      do_something_if_X_is_none_of_the_above();
endselect;
```

Note: In contrast to the "C" programming language, there is no 'break' statement required at the end of each case-block. The program doesn't "fall through" from one case to the next, with one exception:

If there are two or more case marks, with ***nothing in between*** (no 'executable instruction') , the first statement after the case-marks is executed.
Here is a (rather braindead) example:

```
select X
  case 1 : // same handler for cases 1, 2 and 3 ...
  case 2 :
  case 3 :
     print("X = One,Two,or Three");
  case 4 :
     print("X = Four");
  else   :
     print("X is less than one, or larger than four");
endselect;
```

If a case-mark shall actually 'do nothing', use the break statement. When used inside a select-endselect block, **break** actually jumps to the next **endselect**. Example:

```
select X
  case 1 :
     break; // do nothing
  case 2 :
     break; // do nothing as well
  case 3 :
     print("X = Three");
  case 4 :
     print("X = Four");
  else   :
     print("X is less than one, or larger than four");
endselect;
```

A simple example for the select-case construct is in the display application 'ScriptTest1.cvt'.
A longer example for the select-case construct is in the display application 'ScriptTest3.cvt'.
Both applications are contained in the programming tool's installer.

8.8 3.9.8 wait_ms .. wait_resume

Waits for "something to happen", while the CPU can perform other tasks (like updating the display).

wait_ms(N)

blocks the normal script execution for the specified number of milliseconds (N) .
While the script is 'waiting', the normal display program runs faster because all CPU time can now be used for the display.
Without waiting in the main loop (endless loop), script and display-update would still run side-by-side, but the script's main loop would consume an unnecessarily large amount of CPU time.
Recommended waiting intervals (for the script's main loop) are 10 to 50 milliseconds.
Do **NOT** call wait_ms() from event handlers and other 'interrupt-alike' functions !
See also: system.timestamp.

wait_resume

lets the blocked script continue, even if the interval specified in the WAIT_MS command has not expired yet.
Because the normal script execution is blocked, the only place where WAIT_RESUME makes sense is an interrupt function, or in an event-callback (future plan).

See also: system.timestamp, Contents, Keyword List, Quick Reference .

9 3.10 Other functions and commands

In addition to the program flow control commands from the previous chapter, the script language also contains functions and commands for special purposes.

Some of them will be explained in this chapter, while others (like the obvious math functions; random; etc) are only mentioned in the keyword list .

See also: Contents , string processing, file I/O functions, CAN bus functions, screen output, system functions, date- and time conversions,
User-defined functions and procedures .

9.1 3.10.1 Timer (in der Script-Sprache)

The command **setTimer** starts a timer in the script language, for example:

```
      var
        tTimer timer1;  // Declare an instance of a timer as a global variable
of type 'tTimer'
      endvar;
         ...
      setTimer( timer1, 200 ); // Start 'timer1' for a 200-millisecond interval,
                               // here without a timer event handler
         ...
```

To check if a timer is expired (i.e. "programmed time is over"), the script can poll the 'expired' flag in the tTimer structure as in the following code snippet:

```
      if ( timer1.expired ) then
        timer1.expired := FALSE;  // clear the 'expired' flag; it will be set
again 200 ms later
        system.beep( 1000, 1 );   // short beep (1000 Hz, 1 times 100 ms
duration)
      endif;
```

Note: As long as the timer isn't explicitly stopped (i.e. timer1.running isn't FALSE), it will keep on setting the 'expired' flag periodically (every 200 milliseconds in the example shown above). Regardless of when exactly the 'expired'-Flag has been cleared by the application (as in the above example), the timer keeps running synchronously in the background.

As optional third parameter, 'setTimer' accepts the address of a **Timer Event Handler**. Any number of timer event handlers can be implemented in the script language, and (if a handler's address is passed to 'setTimer') the handler will be invoked periodically. Details about that in the chapter about Timer-Events.

9.2 3.10.2 print, gotoxy, cls & Co (output into a multi-line text panel)

The following commands can be used to display text on a multi-line text panel (on any of the programmable display pages, using a "\panel" element in the *display page definition*).

`cls` : "clear screen"

Here: Clears the contents of the text "screen" buffer. Precisely, the buffer is filled with space characters, and all cells are set to the current foreground- and background colour. The script doesn't have direct access to the (graphic) video RAM.

`clreol` : "clear to end of line"

Clears the rest of the current text buffer line, beginning at the current output cursor position. The cursor position itself is ***not*** affected by this command.

`setcolor(foreground,background)` : Set the drawing colours for following text output into the text buffer.

Sets the colour for subsequent calls of print, cls, and clreol. You should not use numeric colour values (because the colour codes may be hardware dependent), but any of the colour-constants listed here .

If any of the two colours shall *not* be modified, pass a negative value as function argument (e.g. -1 = "don't modify").

`RGB`(red component, green component, blue component) : Function to compose a colour value (integer number) from red, green, and blue components.

Besides the colour constants (like clWhite), this is the only recommended method to define colours in your script program.

The value range for each of the three colour components is 0 to 255, regardless of the display's actual number of "bits per pixel". Depending on the display's colour model, not all of the 2 ^ 24 possible colour combinations can be exactly realized ! In such cases, the firmware will try to pick the 'best possible' colour.

Don't make any assumption about the format of the colour number returned by the RGB

function - it's hardware dependent !
The 'Loop Test' application uses the RGB function to produce a colour pattern in a text panel.

`gotoxy(x,y)` : sets the text output cursor into column 'x', line 'y', where x must be an integer value between 0 and 79, and y between 0 and 24 .
The first argument is the zero-based 'X' coordinate (text column), the second argument ('Y') is the text line number.
For example, gotoxy(0,0) will place the text output cursor in the upper left corner of the text screen.

`print` : prints a list of values (numeric and/or strings) to the text screen.
The output cursor position (X) will be incremented for each printed character. If the cursor reaches the end of a line, it wraps to the next.
To 'print' more than one value in a single call, separate the values in the argument list with commas, as in this example:

```
print("\nResults A,B,C = ",A," ",B," ",C)
```

(Remember: backslash-n in a string constant means "new line").
For numeric values (integer or float), print always uses the shortest possible notation, without leading zeroes. If you want fixed lengths, or leading zeroes (as in date and time displayed on the screen), use the itoa function (integer-to-ascii) to convert the numbers into strings with leading zeroes and a fixed width. Example (from 'TimeTest.cvt', shows a calendar date in ISO 8601 format) :

```
print( itoa(year,4), "-", itoa(month,2), "-", itoa(day,2) );
```

`tscreen` : wrapper object for the 'text screen buffer'.

`tscreen.cell[Y][X]`
accesses the character cell in the Y-th line, and X-th column of the text screen buffer as a tScreenCell structure.
Most built-in fonts use DOS-compatible character sets from 'codepage 437' so the text screen has limited graphic capabilities - see TScreenTest example !

`tscreen.cx`
returns the text output cursor's current 'x' coordinate (zero-based column index).

`tscreen.cy`
returns the text output cursor's current 'y' coordinate (zero-based line index).
tscreen.cx and cy are read-only. To modifiy the cursor position, use gotoxy(column,line) .

`tscreen.cs`
Cursor Shape / Cursor Style. Defines *if* and *how* the text output cursor shall be displayed.
The script can use a bitwise combination of the following constants for this purpose:
csOff (Cursor off; default), csUnderscore, csSolidBlock, csBlinking.
The VT100 Emulator example uses this feature to emulate an old-fashioned virtual terminal's cursor display.

`tscreen.xmax`

returns the max. allowed 'x' coordinate of the text buffer (column index).
Since 2013-03-20, tscreen.xmax is also writeable - see details in tscreen.ymax !
The default value of tscreen.xmax is 79 (i.e. 80 characters per line, indexed 0..79).

`tscreen.ymax`

returns the max. allowed 'y' coordinate of the text buffer (line index).
In the first implementation, tscreen.xmax was 79, and tscreen.ymax was 39; but since 2013-03-20, the 'geometry' of the text screen buffer can be adjusted (within certain limits) by assigning new values to tscreen.xmax (first) and tscreen.ymax (second). Most devices are limited to the following values:

- tscreen.xmax must not exceed 99 (i.e. up to **100** characters per line)
- The product of (tscreen.xmax+1) * (tscreen.ymax+1) must not exceed 8000 (because the text screen buffer was limited to 8000 tScreenCell elements in 2013-03-20)

When modifying the 'geometry', first set the lower of the two dimensions (usually tscreen.xmax), so the product never exceeds the maximum. Example:

```
  tscreen.xmax := 39; // only need 40 characters per line
(index 0..39), but..
  tscreen.ymax := 99; // 100 lines (0..99) in the text
screen buffer !
```

Note that (unlike the two functions below), tscreen.xmax and tscreen.ymax do ***not*** depend on the visible text panel element on the current display page .

`tscreen.auto_scroll`

This flag can be set to TRUE by the display application to enable automatic scrolling of the text screen buffer, whenever **tscreen.cy** exceeds **tscreen.ymax** .
By default, automatic scrolling is *disabled* (tscreen.auto_scroll := FALSE), which causes excessive lines to get lost (not printed into the text buffer at all).
An example for an automatically scrolling text panel is in the Internet demo application.

`tscreen.vis_width`

Returns the currently visible width (in characters) of the text screen, which depends on the size, borders, and font of the first visible text panel element ***on the current display page***. For example, if the text panel is 320 pixels wide (without borders), and uses an 8-pixel wide fixed font, the visible width of the text screen will be 40 characters.

`tscreen.vis_height`

Returns the currently visible width (in characters) of the text screen, which depends on the size, borders, and font of the first visible text panel element ***on the current display page***. For example, if the text panel is 240 pixels high (without borders), and uses a 16-pixel wide fixed font, the visible width of the text screen will be 15 characters.
The 'QuadBlocks' demo uses this function to automatically adjust the size of the 'playing field' (actually a text panel) to the size of the screen, if the application (which was designed for a 320*240 pixel screen) is loaded into a device with 480*272 pixels.

`tscreen.scroll_pos_x,`
`tscreen.scroll_pos_y`

Contains the current horizontal and vertical scrolling position for the display of the 'virtual' text screen on a text panel.
With tscreen.scroll_pos_x=0 and tscreen.scroll_pos_y=0, the upper left corner of the text panel will show the character in the first column (x=0), and the first line (y=0) of the virtual text screen (text buffer). An example for the usage of the scroll position can be found in the application 'ScriptTest3.cvt', function 'ScrollIntoView'. By calling the user defined function 'ScrollIntoView' in that demo, the last line 'printed' into the text buffer is made visible, by bringing the scroll-position close enough to the current cursor position. By virtue of a bargraph with 'write access', the vertical scroll indicator can even be used as an interactive control element to scroll the text manually (as far as the size of the screen buffer permits).

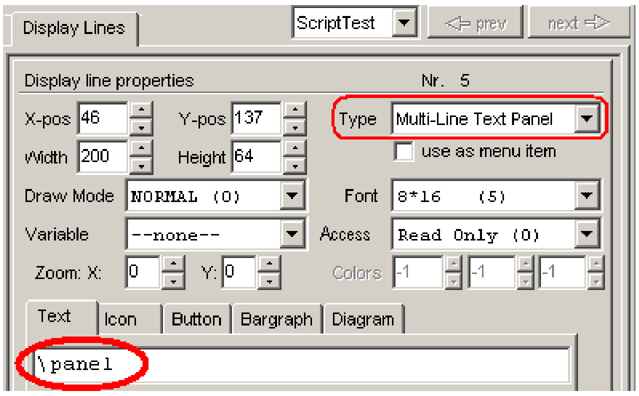
`tscreen.modified`

is TRUE as long as the screen buffer has been modified, but not updated on the LCD yet.
The script can set it (tscreen.modified := TRUE) to let the system redraw the (text-) screen as soon as possible, for example after modifying the screen buffer with the tscreen.cell property. The system will automatically clear this flag when a text-screen-update is 'done'. This function is also used in the TScreenTest example to force an update of the screen, and to find out if (and when) the screen has been updated.

Note:

The output will be 'printed' into the text buffer for a multi-line text panel. If no such panel is visible on the current display page, you will not see it on the screen immedately. But despite that, the text will become visible (immediately without further print-calls) as soon as the display program switches to a page which contains a 'Multi-Line Text Panel'.
In the programming tool, the contents of the text screen buffer can be displayed in der right half of the *Script* tab. In the combo box in the upper right corner, select **'Show buffer for Text Panels'** instead of **'Hide debug view'**.

To create such a text panel (in the definition of one of your display pages), enter the backslash sequence panel in the format string column on the display page definition tab, or change the type of an already existing display line from 'Text' (which means a single-line text display) to 'Multi-Line Text Panel'.



ScriptTest
Nr. 5
X-pos 46 Y-pos 137 Type Multi-Line Text Panel
Width 200 Height 64 use as menu item
Draw Mode NORMAL (0) Font 8*16 (5)
Variable --none-- Access Read Only (0)
Zoom: X: 0 Y: 0 Colors -1 -1 -1
\panel

Don't forget to make the size of the 'Multi-Line Text Panel' large enough ! The screenshot above shows the definition of such a text-panel, taken from one of the 'Script Demo'-applications in the programming tool's programs folder. For example, if the panel 's graphic area is 200 pixels wide and 64 pixels high, and uses an 8 by 16 pixel font, the panel may show up to 25 characters per line (200/8) and four (64/16) lines of text. Regardless of the actual *panel size* (which may be different on each display page), the background text buffer can store a maximum of 40 lines with 80 characters per line. Since these limits may depend on the hardware (possibly larger screens in future), the script can poll the maximum allowed indices into tscreen.cell[Y][X] through tscreen.ymax and tscreen.xmax .

The colour of the character cells inside the text panel is controlled by the script, not by the display page definition. Each character can have its unique foreground- and background colour. The sample application 'LoopTest' contains an example which uses foreground- and background colours composed with the RGB function. The example 'TScreenTest' uses the text array as the 'playfield' for a simple video game ('snake' moving on the screen, controlled via cursor keys), which requires read- and write-access to the characters in the text buffer, without modifying the colours.

See also: multi-line text panel in the display page definitions (external link, only works in the HTML-based help system, not in a PDF document).

9.3 3.10.3 File I/O functions

... are only implemented on systems with a suitable hardware - not necessarily a memory card slot, because the file I/O functions can also access other media (for example, a ramdisk in some devices, or a part of the built-in data FLASH memory in other devices). All file access functions begin with the keyword 'file.' (to avoid namespace pollution). The file I/O functions in the script language may have to be unlocked before use (at least on devices like MKT-View II).

File I/O function overview (follow the links for details) :

- file.create(name, max_size_in_bytes) : creates a new file with the specified name (for write access)
- file.open(name, o_flags) : opens an existing file (only for read access)
- file.write(handle, data) : writes data to a file
- file.read(handle, destination_variable) : reads data from a file (usually 'binary')
- file.read_line(handle) : reads a line of characters from a text file, and returns it as a string
- file.seek(handle, offset, fromwhere) : sets the file pointer
- file.eof(handle) : checks for end-of-file
- file.close(handle) : closes a file, and releases the *file handle* .
- file.size(handle) : returns the size of an *opened* file, measured in byte.

Don't miss the notes on the pseudo-file-system about restrictions of writing files, and how to simulate the pseudo-file-system in the programming tool.
See also: 'File Test' demo (uses most of the file I/O-functions listed above).

3.10.2.1 Pseudo-directories ("folders") in the programmable device

Most of MKT's programmable devices don't really support subdirectories or "folders". Many devices don't even have a memory card interface built inside. Despite that, some of the internal memory (FLASH ROM and/or RAM) can be accessed like a file storage medium ("disk volume").

The principle of pseudo directories is the same as used for the file transfer. In fact, files created this way can be accessed through the file transfer utility or via embedded web server :

- "**`font_flash`**"
  This is another onboad FLASH memory chip (not a FLASH memory card) used to store user defined fonts (*.fnt), but it can be used to store *a few* other files, too. When creating a file in this directory, the maximum expected size must be specified in the 2nd argument of the file.create function. When closing the file again, unused FLASH sectors will be available for other files again. This also applies to the other "..._flash"-folders listed here.
- "**`audio_flash`**"
  Yet another onboard FLASH memory chip (not a FLASH memory card) used to store audio files (*.wav), but it can also be used to store *a few* other files.
  Note: In a few devices which support audio output, but don't have an extra FLASH chip to store the digitized audio, the contents of the 'font_flash' and 'audio_flash' directory may physically be the same. Files placed in this directory can be played back using the interpreter command 'audio.play' .
- "**`data_flash`**"
  This is an internal FLASH memory chip (not a FLASH memory card) used for internal data storage (display pages, imported bitmaps, etc).
  Except for an *upload* of a single *.upt or *.cvt file (via a modified YMODEM-protocol), this directory is not accessable.
- "**`memory_card`**"
  This pseudo-directory can be used to access the removable FLASH memory card, usually an SD-card. If this entry is missing in the pseudo root directory, the device (or firmware) doesn't support such a storage medium. The contents of the 'memory_card' folder will be empty if no card is inserted, or the card's file system is not supported.
  In the programming tool, the device 'memory_card' is simulated by default as a subdirectory named 'sim_mc' (simulated memory card) on the local harddisk. The path to that device is configurable.
- "**`ramdisk`**"
  This pseudo-directory can be used to store *temporary* files, for example bitmap files which may be displayed on the screen without permanently saving them in FLASH memory. When creating a file in this directory, the maximum expected size must be specified in the file.create function. The "File Test" demo uses this pseudo-disk-drive to create a text file, write a few lines into it, close it, open it for reading, and read back the lines which had previously been written. Note that all files in the 'ramdisk' get lost when the device is turned off, or switched into power-down mode.

In addition to the ***storage media*** listed above, the following ***devices*** may be accessed like files (whether they exist depends on the hardware):

- "**`serial1`**"
  Allows accessing the device's first serial port (RS-232) like a file; at least for read- and write operations.
  In rare cases, the script may need to change the serial port settings when opening it. This can be achieved by appending a string with the port settings after the pseudo-devicename, for example:
  `hSerial1 := file.open("serial1/9600");` // try to open 1st serial port, and

configure it for 9600 bits/second .
Other examples for accessing the serial port(s) from the script language can be found here ("GPS Simulator").
Like all other 'extended' script functions, accessing the serial ports (through the file I/O functions) is only possible if the 'extended script functions' (auf deutsch: "Erweiterte Script-Funktionen") have been unlocked !

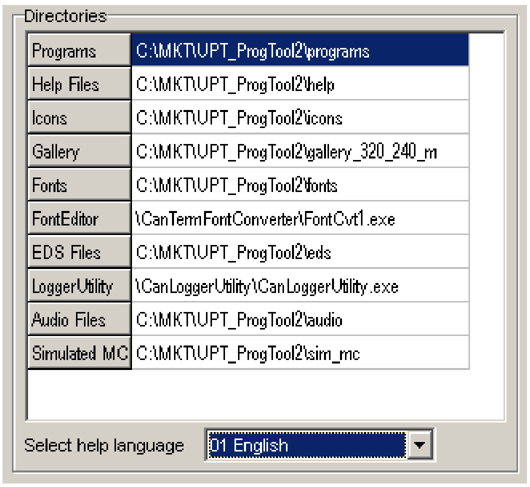
- "**serial2**"
  Similar for the second serial port (if such a port exists, otherwise file.open("serial2") will return zero, which is an illegal handle value.
  An example for accessing this particular port from the script language can be found here ("GPS Simulator").
  In the 'MKT-View II', the second serial port is a dedicated port for a GPS receiver.
  This is not a standard RS-232 port ! Do not try to connected it to the PC with a standard 9-pole "Null-modem cable" ! You may damage your PC, because the 9-pole GPS connector feeds the supply voltage into the external GPS receiver ! Refer to the hardware manual for details, or (if you are unable to find the hardware manual), look here (pinouts of a few 'serial port' connectors).

Important Notes on the Pseudo File System (PFS)

The PFS is not a normal file system (not FAT, NTFS, ext2,3,...) . Except for the "memory_card" device, each file is stored as ***single contiguous block*** on the storage medium, so they can be *directly* accessed via pointer by the CPU - without the need to copy them, sector by sector (at least for read access).
For that reason, certain operations (which you may know from a 'normal' file system) are impossible here:

- For FLASH files, the expected file size must be specified upon creation .
  It may be closed with a 'shorter' size later, but the file cannot 'grow' larger than the pre-allocated size while writing.
- Write-operation is only possible for 'new' files (i.e. after file.create, not after file.open)
- Deleting a single file may be possible, but due to the large FLASH sector sizes (64 kByte or even 128 kByte), the space occupied by the file cannot be freed, because there may be multiple files stored in a single FLASH sector (much in contrast to a normal file system, where each file occupies at least one (disk-) sector of 512 bytes).
- Deleting a single file in a RAMDISK cannot move the other files in memory (because other files may be accessed via pointer for reading, as explained above). Deleting a single file in a RAMDISK (without re-formatting the RAMDISK) is only possible if that file is still the last file written to the disk.
  For this reason, delete temporary files which your script may have created on the ramdisk *as soon as possible*. Deleting it 'too late' will not free the memory.
- Remember that the RAMDISK is not battery-buffered: It will automatically be reformatted whenever the system is booted, and all files in it will be lost !

To simulate the various STORAGE MEDIA in the programming tool, real 'disk files' are used. The directories ("folders") used for those files can be configured ***on the 'Settings' tab*** in the programming tool. Double-click into the table to open a file selector if you need to change these entries:



Directories

Programs	C:\MKT\UPT_ProgTool2\programs
Help Files	C:\MKT\UPT_ProgTool2\help
Icons	C:\MKT\UPT_ProgTool2\icons
Gallery	C:\MKT\UPT_ProgTool2\gallery_320_240_m
Fonts	C:\MKT\UPT_ProgTool2\fonts
FontEditor	\CanTermFontConverter\FontCvt1.exe
EDS Files	C:\MKT\UPT_ProgTool2\eds
LoggerUtility	\CanLoggerUtility\CanLoggerUtility.exe
Audio Files	C:\MKT\UPT_ProgTool2\audio
Simulated MC	C:\MKT\UPT_ProgTool2\sim_mc

Select help language: 01 English

For example, to access files in the "audio_flash" folder, copy the required files into that directory, or change the directory location (inside the programming tool) to the place on your harddisk where the program tool can find the files.
It is very advisable that while developing your application, you ***make a list of all files which your application may need later*** (during runtime on the "real target"). Such files may include (but are not limited to) the following:

- The display application itself (*.cvt or *.upt)
- user-defined **fonts** (*.fnt)
- **bitmaps** (*.bmp) which your application may expect in any of the pseudo-file folders (not the "normal" bitmaps, which are imported in the programming tool, because those icons will be saved and transferred along with your *.cvt or *.upt file)
- **audio files** (*.wav)
- **text files** (*.txt) in different languages. For example, see the 'MultiLanguageTest': In that application, the script reads the texts from an external file, to separate the display program and the displayed text strings.

Remember to backup and distribute all those files along with your application !

3.10.2.2 Creating or opening a file

file.create(name, max_size_in_bytes) : creates a new file with the specified name (for write access)
If the file already exists, it will be overwritten. If it doesn't exist, it will be created, with an initial length of zero.
The second parameter ('maximum size in bytes') is used for files in the RAMDISK, to pre-allocate the maximum required file size in advance. This avoids fragmentation, if multiple writeable files are opened simultaneously. When successfull, the function returns a positive 'handle' (= an integer value which identifies the file).
Otherwise, the function returns a negative error code.
The filename may contain a pseudo-directory to specify the storage medium.
The parameter 'max_size_in_bytes' (maximum expected size of the file, measured in byte)

must already be specified when creating the file, to avoid fragmentation of the storage medium (see notes about the RAMDISK).
Example to create and write a small file in the RAMDISK, with minimum error checking :

```
  var
    int fh; // file handle
  endvar;
  fh := file.create("ramdisk/test.txt",4096); // max 4096
bytes
  if( fh>0 ) then   // successfully created the file ?
    file.write(fh,"First line in the test file.\r\n");
    file.close(fh); // never forget to close files !
  endif;
```

file.open(name [, o_flags]) : opens an *existing file* or a *device*

When successful, the function returns a positive 'handle' (= an integer value which identifies the file or device).
Otherwise, the function returns a negative error code.
Depending on the storage medium, some (not all, depending on the medium) of the following optional 'open flags' are supported:

- O_RDONLY : Open for read-only, i.e. the file can only be read but not written. Works on all media.
- O_WRONLY : Open for write-only, i.e. the file can only be written but not read. Not supported yet.
- O_RDWR : Open for read- and write access. At the moment (2011-09), doesn't work with FLASH memory !
- O_TEXT : Open the file as a text file, and check for a BOM (byte order mark) to find out the encoding-type automatically.

Regardless of the file-open mode (O_RDONLY, O_WRONLY, O_RDWR, O_TEXT), the file pointer will be set to the *begin* of the file. This behavious equals the _rtl_open command, which may be familiar for 'C' developers. To *append new data at the end* of an existing file (after re-opening it), the file pointer must be set to the end of the file, before writing data to it. Use the function file.seek for this purpose.

Example to open a text file, read it line-by-line, and dump the lines to the screen:

```
  var            // declare global variables:
    int fh;      // an integer for the file handle
    string temp; // a string named 'temp'
  endvar;
  fh := file.open("ramdisk/test.txt",O_RDONLY);
  if( fh>0 ) then // successfully created the file ?
    while( ! file.eof(fh) )
      temp := file.read_line(fh);
      print( "\r\n ", temp ); // dump the line to the screen
    endwhile;
    file.close(fh);
  endif;
```

3.10.2.3 Writing to, and reading from files

After successfully creating or opening a file, the file's *handle* (integer value returned by file.create or file.open) can be used to write to, or read from the file.
The following script functions can be used for this purpose.

file.write(handle, data): writes data to a file

In most cases, 'data' will be a string, one or more a string variables, or a string expressions.
(In fact, binary data are not supported yet, because they always turn into a nightmare because you need to worry about the host's endianness aka "byte order", different storage formats for all kinds of data types, etc etc - so forget about binary files for a while).
An example for the file.write function can be found under file.create .

file.read(handle, destination_variable) : reads data from a file (usually 'binary')

... since binary files are not supported yet, better use file.read_line which makes processing text files easier ...

file.read_line(handle) : reads a line of characters from a text file (or a serial port), and returns it as a string

An example using 'file.read_line' can be found under file.open .
To read the lines of a text file line-by-line, you should open the file with the O_TEXT flag as explained in the file.open command,
because strings read from a file read that way will have the proper character encoding type (ceDOS, ceANSI, or ceUnicode).

file.eof(handle) : checks for end-of-file

Returns FALSE (0) as long as the end of the file has not been reached yet,
and TRUE (1) when the end-of-file has been reached (for example, after file.read_line has read the last line of a file).
An example using 'file.eof' can be found under file.open .

file.seek(handle, offset, fromwhere) : sets the file pointer.

'offset' is the absolute or relative file position, measured in bytes.
'fromwhere' (aka 'origin') specifies the meaning of 'offset'. This parameter may be one of the following constants:

- **`SEEK_SET`** Position file relative to the beginning (offset 0 = first byte in the file)
- **`SEEK_CUR`** Position file relative to the current position
- **`SEEK_END`** Position file relative to the end (offset 0 = last byte in the file)

The value returned by file.seek indicates the new, absolute file pointer, measured in bytes from the beginning.
For 'SEEK_END', the offset may be zero (i.e. warp to the end of the file) or *negative* !
Positive offsets in combination with 'SEEK_END' would reference a position past the file's end, which is illegal.
An example using 'file.seek' is in the 'File Test' demo.

file.close(handle) : closes a file, and releases the *file handle* .

An example for the file.close function can be found under file.create .
Never forget to close files ! Especially after a file has been 'written' but not been closed yet, there may be data waiting in an internal buffer which have not been flushed to disk yet. Closing the file will flush all buffers, and (if it's a "real disk file") update the directory and the FAT (file allocation table).

By default, text files are assumed to be 'plain text with 8 bits per character'. Esoteric formats like *.doc or *.docx are not, and never will be, supported. When specficying the O_TEXT flag in file.open(), the runtime library will examine the file for a so-called Byte Order Mark (BOM), to find out if the file's encoding types automatically (and skip the BOM, so the BOM will not be read by the first call of file.read_line) :

- UTF-16 with big-endian byte order (BOM = 0xFE, 0xFF)
- UTF-16 with little-endian byte order (BOM = 0xFF, 0xFE)
- UTF-8 (BOM = 0xEF, 0xBB, 0xBF, not really a byte order mark in this case)

If the file contains Unicode text (in one of the encodings listed above), the strings returned by the file.read_line function will be re-encoded as UTF-8 (not UTF-16 !).
For more info about the byte order mark in text files, see Byte order mark on Wikipedia.

See also: Overview of file I/O functions, pseudo file system, accessing files via web server, string processing, string processing, keywords, contents .

9.4 3.10.4 Reception and Transmission of CAN messages (via script)

Note: Not all programmable terminals support the reception of 'raw' CAN messages as explained in this chapter.
Devices with CANopen do *not* support the functions mentioned below.
The CAN functions in the script language may have to be unlocked before use (on devices like MKT-View II).

Use the command can_add_id or can_add_id_filter to add some CAN-bus identifiers for reception through the can_receive function (or via CAN-receive-handler).
Then, use the can_receive function to check for messages waiting in the FIFO, and to read the next message from the FIFO.
The received CAN message (aka "telegram") will be copied (by can_receive) from the FIFO into a global variable (structure) named can_rx_msg .

As long as the script doesn't call can_receive again, the contents of can_rx_msg will not change, so the script can directly process the contents of can_rx_msg in any way.

3.10.4.1 **can_add_id** (procedure)

Adds the specified CAN message identifier to an internal list (in the CAN driver), so it will be received from now on, and put in the script's CAN receive FIFO.
The script program only receives such messages with the can_receive function. CAN-Messages which are *not* registered for reception this way may be processed somewhere else (for example in the "CAN-signal-decoder" or in the CANopen stack, but not in the script).
The maximum number of CAN messages identifiers which can be registered *for reception by the script* is 32 (since 2011-05-05).

Since 2013-05, can_add_id accepts an optional second parameter, to specify the name of a CAN-Receive-Handler (which can be written in the script language). Example:
can_add_id(0x123, CAN_Handler_ID123); // Call 'CAN_Handler_ID123' on reception of this message ID

Details about CAN-receive handlers in chapter 3.11.4. (CAN-Receive Handler).

3.10.4.2 **CAN.add_filter**(<filter>, <mask>, <receive_handler>)

Similar to can_add_id, but this command registers an entire **range** of CAN message identifiers for reception.

The CAN-receive-filter operates as follows:

The CAN identifier of a received message is bitwise ANDed with the 'mask' parameter.
The result is then compared with the 'filter' parameter.
If all bits (which are not zero in 'mask') match, the received message will be passed on to the script (either to the optional receive-handler, or placed in the script's CAN-receive-Fifo).
In other words: All **cleared** bits in 'mask' are considered 'don't care' (i.e. ignored by the filter); all bits **set** in 'mask' must match (between received ID and the 'filter' value).

Example:

```
CAN.add_filter( 0x2ABCD00, 0x2FFFFF00 ); // receive extended IDs 0x0ABCD00 to
0x0ABCDFF
// Note: As in other CAN functions of the script language,
// the 'extended' flag which indicates a 29-bit-ID is encoded in bit 29,
// and the bus number (0..3) is encoded as a two-bit value in bits 31..30 of the ID.
```

Only **one** range of CAN-message-identifiers can be registered per interface, in addition to the 'individually' registered (single) message identifiers from can_add_id.
This function had to be implemented because J1939 encodes a lot of information (for example the sender's 'Source Address', SA) inside the 29-bit CAN message ID.
This renders the CAN acceptance filtering, which is implemented 'in silicon' (hardware) in most microcontrollers, almost useless because any non-trivial J1939 node is doomed(!) to receive 'almost everything' this way.
The result from registering 'all CAN message identifiers' for reception may cause an enormous CPU load, so if you don't need it (to implement J1939 in your scripts), don't use **can_add_id_filter** at all.

3.10.4.3 **can_receive** (function)

Tries to read the next received CAN message from a FIFO.
When successful, the message is copied into can_rx_msg, and the result is 1 (one) .
Otherwise (empty FIFO), can_rx_msg remains unchanged, and the result is 0 (zero) .

3.10.4.4 **can_rx_fifo_usage** (function)

Returns the number of CAN messages still waiting in the receive FIFO (without reading a message).
A return value of zero means "the FIFO is completely empty at the moment".
A return value of 2047 (!) means "the FIFO is completely full" (and, if another message was received by the CAN controller, it would be lost for the script).

3.10.4.5 **can_transmit** (procedure)

Command to send a CAN message (directly, layer 2). This command comes in two different variants (without and with a parameter in the argument list):

Variant 1: **can_transmit without an argument list**:
Tries to send the contents of can_tx_msg (CAN message structure defined below). The global variable 'can_tx_msg' is filled with contents by the script, and transmitted my calling can_transmit:

```
   can_tx_msg.id  := 0x334;  // set CAN message ID (and bus number in the upper bits)
   can_tx_msg.len := 2;      // set the data length code (number of data bytes)
   can_tx_msg.b[0] := 0x11;  // set the first data byte
   can_tx_msg.b[1] := 0x22;  // set the second data byte
   can_transmit;  // send the contents of can_tx_msg to the CAN bus
```

Variant 2: **can_transmit called with a parameter (argument)**:
Instead of using the global variable 'can_tx_msg', a variable (preferrably a *local* variable) of type 'tCANmsg' is filled by the script, and the address of that variable is passed as an argument to the procedure 'can_transmit(<address of the message to be sent>) .
The following example calls can_transmit (with parameter) from a CAN-receive handler, after assembling the transmitted message in a *local* variable ("responseMsg"):

```
 //----------------------------------------------------------------------
 func CAN_Handler_A( tCANmsg ptr pRcvdCANmsg )
   // A CAN-Receive-Handler for a certain CAN message identifier.
   // Must be registered via 'can_add_id', along with the CAN message ID.
   // Interrupts the normal script processing, and must RETURN to the caller
   //  a.s.a.p. ! Uses a LOCAL variable for transmission (not can_tx_msg).
   // Thus can_tx_msg can safely be used in the script's main loop,
   // even if the main loop may be interrupted at any time by this handler.
   local tCANmsg responseMsg;    // a local variable with type 'CAN message'
   responseMsg := pRcvdCANmsg[0];      // copy the received CAN message into the response
   responseMsg.id := pRcvdCANmsg.id+1; // response CAN ID := received CAN ID + 1
          // Note: The upper two bits in tCANmsg.id contain the zero-based BUS NUMBER !
   can_transmit( responseMsg );  // send a response via CAN immediately
   return TRUE;
   // returning TRUE means: "the received message was processed HERE,
   //              do NOT place it in the script's CAN-receive-FIFO".
 endfunc; // end CAN_Handler_A

     ... somewhere in the initialisation : ...

   // Register a received CAN ID, and install a CAN-receive-handler for it:
   can_add_id( C_CANID_RX_A, addr(CAN_Handler_A) );
```

An example for the *periodic* transmission of CAN messages is in the application 'TimerEvents.cvt' (look for "OnCANtxTimer").

Please note that this procedure usually returns 'immediately', before the transmission actually took place, because all transmitted CAN messages (not just those sent from the script) are first placed in an interrupt-driven CAN transmit buffer. This is the reason why `can_transmit` cannot return a "status" (like 'transmission complete', etc). To send RTR frames (Remote Transmit Request), bitwise-OR the constant cCanRTR in the length field of the message.

Only use this function if you know exactly what you are doing !
Sending a 'wrong' CAN message into an unknown network may have potentially dangerous consequences !

3.10.4.6 **can_rx_msg** , **can_tx_msg** (global variables)

From the script's point of view, the 'can_rx_msg' structure holds the last received CAN message for processing. It was filled by the previous call of the can_receive function. If the script doesn't call can_receive, and as long as can_receive doesn't return TRUE (=success), the contents of can_rx_msg will not change. The following components of can_rx_msg (and similar, can_tx_msg for transmission) can be accessed like variables by the script program:

.id

holds the CAN-bus-identifier in the least significant 11 (or 29) bits of this 32-bit integer variable, plus an optional 11/29-bit flag ("extended CAN identifier flag") in bit 29, and the zero-based CAN BUS NUMBER (!) in the most significant bits (bits 31..30). Please note that bit numbers are always start at zero, thus a 32-bit integer value (such as can_rx_msg.id) contains bit 0 (least significant bit) to 31 (most significant bit). There is no "bit 32" in a 32-bit integer !

.len

Length of the DATA FIELD in bytes. CAN messages can have 0 (zero) to 8 byte data fields. The upper bits of this field *may* contain special FLAGS like cCanRTR (Remote Transmission Request, see example ScriptTest3.cvt, SendRTR()).

.tim

Timestamp of the received CAN message, using the CAN driver's hardware specific timestamp frequency. This 32-bit integer value will roll over from 0xFFFFFFFF to 0x00000000 after about 29 hours, because the CAN driver's timestamp clock frequency is 40 kHz (at least for the ARM-7 CPUs with an internal CPU clock of 72 MHz). If you *really* need to convert a *timestamp difference (as 32-bit integer value)* into seconds or similar, divide the difference by the constant 'cTimestampFrequency' to get a timestamp difference in seconds. Note that other script timer functions use the same 'timestamp'. See also: system.timestamp ("current time", using the same unit) .

.b[N]

Accesses the N-th byte in the CAN data field as an 8-bit unsigned integer (value range 0 to 255).

.dw[N]

Accesses the N-th **Doubleword** (N : 0..1) in the CAN data field. A doubleword ('DWORD') contains 32 bits, the value range is 0x00000000 to 0xFFFFFFFF (hex).
Note that (in contrast to the components listed further below) the array-index is a 'doubleword'-index, not a byte-index. Thus the only allowed indices are 0 (=first doubleword) and 1 (=second doubleword).
Example to copy the entire 8-byte CAN data field (taken from 'ScriptTest3.CVT') :

```
// The 8 databytes are copied as two 32-bit integers,
// because that's faster on an ARM-CPU than a byte-copying-loop:
response.dw[0] := can_rx_msg.dw[0]; // copy first doubleword (4 bytes)
response.dw[1] := can_rx_msg.dw[1]; // copy second doubleword (4 bytes)
```

Unlike the components of 'can_rx_msg' and 'can_tx_msg' listed further below, the DWORD-wise access explained above is also possible for 'normal' varibles declared as type tCANmsg (also via pointer).

.i16[N]

Accesses the N-th and N+1-th byte in the CAN data field as a 16-bit ***signed*** integer, using 'Intel' byte order (aka Little-Endian, or 'least significant byte first).
The sign is expanded from bit 15 into the upper bits of the 32-bit integer result, so the range is -32768 to +32767.

.u16[N]

Accesses the N-th and N+1-th byte in the CAN data field as a 16-bit ***unsigned*** integer, using 'Intel' byte order (aka Little-Endian, or 'least significant byte first). Unlike 'i16', the sign bit is not expanded, so the result ranges from 0 to 65536.

.i32[N]

Accesses the N-th to N+3-th byte in the CAN data field as a 32-bit signed integer, using 'Intel' byte order (aka Little-Endian, or 'least significant byte first).
Note that the above three methods to access a CAN data field are very fast, but they cannot cross 'arbitrary' bit boundaries (thus, their syntax mimicks a BYTE-ARRAY, not a BIT-ARRAY). The "bitfield" method explained further below is slower, but more versatile.

.m16[N]

Similar to '.i16', but uses big endian byte order aka 'Motorola' format.

.m32[N]

Similar to '.i32', but uses big endian byte order aka 'Motorola' format.

.bitfield[<index of the signal's least significant bit in the CAN-message> **,** <number of bits> **]**

Accesses a part of the data field in a CAN messages as a 'bitfield', containing an *unsigned* integer value in 'Intel' byte order (least significant byte first).
Note that regardless of the signal type, bits in a CAN message are always numbered according to the following table. As usual for binary numbers, the most significant databit is on the left side *in this graphic representation*; the numbers for 8 bits within a byte always runs from 0

(=LSB, right) to 7 (=MSB, left).
The green cells in the following table show an example defined as
`can_rx_msg.bitfield[ 18, 15 ]`
("Intel" byte order, LSBit at bit 18 in the CAN frame, and 15 bits for this bitfield).
Support for signals with 'Motorola' byte order (most significant byte first) was not projected by the time of this writing.

Table X : Numbering of bits in a CAN data field
(**green**: 15-bit signal example; .bitfield[18, 15])

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
byte[0]	7	6	5	4	3	2	1	0
byte[1]	15	14	13	12	13	10	9	8
byte[2]	23	22	21	20	19	18	17	16
byte[3]	31	30	29	28	27	26	25	24
byte[4]	39	38	37	36	35	34	33	32
byte[5]	47	46	45	44	43	42	41	40
byte[6]	55	54	53	52	51	50	49	48
byte[7]	63	62	61	60	59	58	57	56

The global variable 'can_tx_msg' has exactly the same structure as 'can_rx_msg' . The only difference is that can_rx_msg is used for reception, while can_tx_msg is used for transmission (from the script's, i.e. the device's, point of view). Typically, both are used in the implementation of a simple 'CAN protocol handler'.

The script test application 'ScriptTest3.cvt' contains a few examples for the reception and transmission of CAN messages through the script interpreter.

Notes and hints:

To test the script's CAN functions in the programming tool, connect your PC to the CAN bus using one of the supported CAN interfaces. The script language's CAN RX FIFO also works in the simulator, using *live data* received from the CAN bus.
If that is not possible (no CAN bus available in the lab, or no suitable CAN interface on the PC), use the programming tool's CAN playback utility. It allows you to play back recorded CAN messages (stored in a simple text file) into the simulator/emulator, as if they were received from a 'real' CAN interface.

A valuable tool for the development of CAN protocols in the script language is the Trace History, which most devices support (also those without an integrated CAN logger / snooper). The history can be read out remotely via web browser, for example using the URL http://upt/trace.htm . In contrast to external CAN diagnostic tools, the trace history display distinguishes between *sent* and *received* CAN messages (from the device's point of view), which an normal external CAN bus monitor can't (from *his* point of view, all CAN messages are *received*).

3.10.4.7 Special CAN-bus diagnostic functions

Most of the following 'special' CAN functions only work on certain targets, but not in the programming tool / simulator . The first function argument is usually the CAN port number. Use one of the following integer constants for the port number:
 `cPortCAN1` (1st CAN bus),
 `cPortCAN2` (2nd CAN bus) .

The application script_demos/ErrFrame.CVT uses some of these functions to test a CAN bus with error frames.

`CAN.rx_counter( <port> )`

Retrieves the number of CAN messages received through the specified port, since power-on.

`CAN.tx_counter( <port> )`

Retrieves the number of CAN messages sent through the specified port, since power-on.

`CAN.err_counter( <port> )`

Retrieves the number of any errors and warnings, counted by the CAN-driver's interrupt service handler since power-on.
The counter includes 'comparably harmless' errors, for example bit-stuffing errors, which sporadically occur even in a 'good' CAN network. This function is only intended for diagnostic purposes; the expectable error rate depends largely on the bus load and on the environment (EMC) !

`CAN.err_register( <port> )`

Retrieves the last content of the CAN controller's 'error register', captured by the CAN-driver's interrupt service handler when the last CAN error interrupt occurred.
Since the format of the CAN controller's error register is extremely hardware dependent, specifying the meaning of the bits in that register would be far beyond the scope of this documentation.

- For MKT-View II (with CPU = LPC2468), see NXP's "UM10237" (LPC24XX User Manual), Chapter 18.8.4, "Interrupt and Capture Register";
- For MKT-View III (with CPU = LPC1788), see NXP's "UM10470" (LPC178x/7x User Manual), Chapter 20.7.4, "Interrupt and Capture Register", pages 514 to 517 in UM10470 Rev. 1.5;
- For devices with other controllers, and in the programming tool (simulator), the value returned by CAN.err_register() is meaningless !

`CAN.err_frame_counter( <port> )`

Retrieves the number of "error frames" received on the specified port, since power-on.
Strictly defined, it's the number of "bit stuffing errors" signalized by the CAN bus controller. A bit-stuffing error means six or more dominant bits on the physical layer.
Ideally, there should be no error frames on a CAN at all. The error-frame-counter allows to check this.

`CAN.PulseOut( <port> , <duration in microseconds> )`

Generates a pulse (dominant state) on the specified CAN port, with the specified length (duration) in microseconds.
Rarely used; for example to send CAN error frames (= six dominant bits, which is impossible with a 'normal' CAN controller.
The function only works on a suitable target (LPC / ARM-7), not on a PC, and not on any Linux-based system. A sample script which uses this procedure is in the 'ErrFrame' application in the script_demos folder. It was designed to *send* CAN error frames, to check if certain CAN testers were able to detect such CAN bus errors. For example, see the ErrFrame.CVT application.
Do not use this function in a critial environment (vehicle, etc), unless you are absolutely sure about the possible consequences !

9.5 3.10.5 Controlling the programmable display pages from the script

Any procedure or function beginning with the keyword "display" controls the *programmable display pages* in some way.
An important function is to pause the display output temporarily, for example to ensure the consistency (auf Deutsch: Widerspruchsfreiheit) of the screen while the script prepares some values which "always belong together". Without such precautions, due to semi-multitasking of the script and the display, some of those values shown on the display may be "old", and others may be "new".

`display.goto_page(` <page> **`)`**

Switches to the specified display page.
Caution: Due to the script's semi-multitasking, the actual page-switch will not happen 'immediately', but at the next time when the *display interpreter* has the chance to update the current display page ! The script's runtime engine doesn't wait until the page-switch has been finished !
The new page can be specified either as a page number (deprecated), or as a page's *name* (favourized).

`display.page_name`

Retrieves the name of the current display page (read-only).

`display.page_index`

Retrieves the zero-based index of the current display page (read-only).
Remember, the "first" page of every display application has index "zero", not "one" !
To switch to a different display page (from the script), use display.goto_page .

`display.num_pages`

Retrieves the number of pages which exist in the display application (read-only).
This function can be used to let the script run through a 'loop with all display-pages', as used in the 'page menu' example.

`display.num_lines`

Retrieves the number of lines on the current display page (read-only).
This function can be used to let the script iterate through 'all elements on the current display page'.

`display.page[n].name`

Retrieves the name of the n-th display page (read-only).
Note that the page index 'n' runs from **zero** to **display.num_pages minus one** !

`display.exec( < command string > )`

Lets the display-interpreter execute any display command. This command must be used *with caution* ! It should be avoided if not absolutely necessary, because it may severely slow down the script. This may happen because the *display* commands are *interpreted*, not compiled .
Example:
`display.exec( "bl(0)" );` // turn the display's backlight off via display interpreter
To avoid calling the display interpreter from the script, use 'flag variables', which can be polled in global or local events by the display. This way, the script will not be slowed down (or even blocked for dozens of milliseconds) by the execution of the display command. A safe example for the display.exec command can be found in the 'traffic light' demo.

`display.pixels_x`

Retrieves the width of the LC display, measured in *pixels* (read-only). Used in the QuadBlocks demo to switch to a display page designed for 'landscape' or 'portrait' mode of the screen.

`display.pixels_y`

Retrieves the height of the LC display, measured in *pixels* (read-only).

`display.fg_color`

Returns the default foreground- aka text-colour of the current display page.
In the programming tool, this colour value is defined in the im 'Display Page Header' (Default Text Colour).

`display.bg_color`

Returns the default background colour of the current display page.
In the programming tool, this colour value is defined in the im 'Display Page Header' (Default Background Colour).

`display.menu_mode` , **`display.menu_index`**

This is the script language's equivalent to the display interpreter's "mm", and "mi" function.
(Entspricht der Funktion "mm" bzw "mi" des Display-Interpreters.)
Examples in the application 'DisplayTest.cvt' .
Possible **menu modes** are defined as built-in constants in the script language:
`mmOff` : neither 'navigating' nor 'editing'
`mmNavigate` : navigating between different fields on the page
`mmEdit` : editing the (usually numeric) value in an input-field

`display.elem[<Element-Name>].visible`

The script can show or hide a display element by setting or clearing the 'visible'-flag.
Example:

```
   display.elem["Arrow"].visible := TRUE;  // show the element
named "Arrow"
   display.elem["Popup"].visible := FALSE; // hide the element
named "Popup"
```

Note: When a display page is loaded from ROM (due to a 'page switch'), all elements are visible by default.
Making an element *invisible* which was previously *visible* causes an automatic update of the entire display page.

`display.elem[<Element-Name>].xyz` or
`display.elem[<Element-Index>].xyz`

This is the *script language*'s equivalent to the *display interpreter*'s function "disp.<Element>.xyz" (follow the link for a list of accessable components, here simpy called 'xyz').
Examples (more in the 'display test' application):

```
// Show the NAME of the currently selected element :
print("\r\n Name:", display.elem[ display.menu_index ].na );

display.elem[i].bc := RGB(255,127,127); // set background to
lightred
```

The element can be addressed by its *index* as in the above example, or by its *name*. Example:

```
display.elem["BtnNext"].bc := RGB(0,i,255-i); // modify 1st
background colour
display.elem["BtnNext"].b2 := RGB(0,255-i,i); // modify 2nd
background colour
```

In the last example, "BtnNext" is the name of an UPT display element, specified in the page definition table. The script runtime determines which of the two addressing modes is used by the *data type* of the argument between the squared brackets: [Integer] = by index, [String] = by name.

`display.elem_by_id[<Control-ID>].xyz`

Similar as the above (display.elem), but accesses a display element by its control-ID (which needs to be defined in the page definition table).
Accessing a display element this way simplifies modifying it in an event handler (in the script language), because the control-ID is passed in the handler's function argument list. Details about this "advanced" topic are here .

`display.pause := TRUE;`

Stops updating the screen (with the current "programmed display page"). This can be used for a crude way of synchronizing the display to the script application: Pause the display before calculating a new set of 'display values' in the script, and resume when finished with that.

~~**`display.dia.XYZ`**~~

~~Invokes one of the commands to control the Y(t) or X/Y-diagram on the current display page. Details are in a separate document about the display interpreter's 'diagram' commands. 'XYZ' is the token listed in that document, for example **clear, run, ready, ch[N].min, ch[N].max, sc.xmin, sc.xmax**, etc.~~

`display.pause := FALSE;`

Resumes updating the screen, i.e. switches back to normal periodic screen update (screen updated 'in the background', *while* the script runs).
The test/demo application "LoopTest.cvt" uses the display.pause flag to avoid flicker, while filling the text-screen with new data.

`display.redraw := TRUE;`

Sets a flag to let the *UPT display interpreter* update the current display page as soon as possible. On completion, the flag display.redraw will be cleared automatically.
It is usually *not* necessary to force a display update this way, because the *display interpreter* periodically compares the values of all (numeric) variables associated with the elements on the current display page; and -if a value has changed since the last update- automatically redraws the element.

See also ... about interaction between *script* and *display application* :

- Accessing ***display variables*** from the script
- Accessing ***script variables*** from the display interpreter
- Invoking ***script procedures*** from the display interpreter
- Invoking ***script functions*** from display pages (to retrieve a text strings for the display, used for internationalisation)
- Asynchronous event handling (and how to *intercept* certain events in the script language)

3.10.6 'System' functions, etc

Built-in procedures or function begin with the keyword "system". They access some low-level system parameters. At the time of this writing (2011-09-29), the following system functions were implemented :

`system.audio_vol`

Reads or writes the audio output volume aka "Speaker Volume". The same parameter can be modified (and permanently saved) in the terminal's system menu. The function is only implemented in certain terminals with an analog audio output - see feature matrix . A similar command also exists in the display interpreter. Unfortunately, the value range and scaling depends on the hardware. For example, the CVT-MIL 320 uses a digital potentiometer with 128 linear steps (not logarithmic!), value range 0 to 127.

`system.beep(` frequency [,time [,volume [,freq_mod [,ampl_mod]]]])

Produce a simple sound using the system's built-in 'beeper' (buzzer, piezo speaker, or similar). Similar as the 'beep' command in the older display interpreter.
frequency : Tone frequency in Hertz. A value of zero turns the tone off.

time: length of the tone, measused in 100-millisecond-steps. If this and all following parameters are missing (or zero), the tone will be "endless" until you turn it off with the command system.beep(0).
volume: Relative volume (loudness) in percent, ranging from 0 to 100. The beeper is controlled with a pulse width modulator which can be used to produce different output levels, but the harmonic spectrum of the generated tone is also affected by the PWM duty cycle. A volume of 100 produces the loudest possible tone with a 1:1 duty cycle. *freq_mod*: Frequency modulation. Can be used to produce siren-like sounds, or "chirps" and "whistles". Unit is "Hertz per 100 milliseconds". If the value is positive, the frequency increases as long as the sound is audible; if the value is negative the frequency decreases.
ampl_mod: Amplitude modulation. Can be used to produce sounds which start with a low volume and then get louder. Not very effective because of the pulse-with modulation, where a volume of 10% can hardly be distinguished from a volume of 50% .
Example: system.beep(150,20,50,100)
produces a 2-second, "chirped" tone which rises from 150 Hz to 2150 Hz (=150 Hz + 2 seconds * 100 Hz/0.1sec)

`system.click_vol`

Only for devices with touchscreen. Reads or writes the 'touchscreen click' volume. The same parameter can be modified (and permanently saved) in the terminal's system menu.

`system.led( index, pattern, red, green, blue )`

Only for devices with multi-colour LEDs. Sets the blink-pattern and RGB colour mixture for the specified LED (as usual, indices begin at zero, not one).
Parameters:

```
  index :  0=first LED (topmost on the MKT-View 2)  ... 2=third LED
  pattern: 8-bit blink pattern. Each bit controls a 100-millisecond
interval.
           After a cycle of 8 * 100 ms the whole cycle starts again.
           The 8-bit blink patterns of all LEDs are synchronized.
  red, green, blue: specifies the 3 * 8-bit colour mixture (see examples).
```

If a certain LED parameter (pattern, red, green, blue) shall *not* be modified, pass -1 (a negative value) instead.
Examples:

```
  system.led( 0, 0xFF, 0xFF, 0x00, 0x00 ); // 1st LED permanently on, RED
  system.led( 1, 0x0F, 0x00, 0xFF, 0x00 ); // 2nd LED slowly blinking GREEN
  system.led( 2, 0x55, 0x3F, 0x3F, 0xFF ); // 3rd LED rapidly flashing BLUE
  system.led( 2, 0x00, -1,   -1,   -1   ); // 3rd LED off without changing
the colour
  system.led( 2, 0xFF, -1,   -1,   -1   ); // 3rd LED on without changing
the colour
```

`system.dwInputs`

Access the system's onboard digital inputs as a 32-bit 'doubleword'. Depending on the target hardware, up to (!) 32 digital inputs can be read in a single access (of course, not all devices have onboard digital I/O lines at all). Bit zero reflects the state of the first input, etc. Use a formal assignment to read the current state of the digital inputs, for example:

```
iDigitalInputs := system.dwInputs; // poll all onboard digital
inputs
if( iDigitalInputs & 0x00000001 ) then // check bit zero =
first input
  print("DigIn1 = high");
else
  print("DigIn1 = low");
endif;
```

`system.dwOutputs`

Access the system's onboard digital outputs as a 32-bit 'doubleword'. Depending on the target hardware, up to (!) 32 digital inputs can be read in a single access (of course, not all devices have onboard digital I/O lines at all, and for most devices, it's impossible to set all digital outputs exactly at the same time, due to hardware restrictions / internal 'I/O-bus'). Bit zero reflects the state of the first input, etc. Use a formal assignment to read, modify, and write the current state of the digital outputs, for example:

```
system.dwOutputs := system.dwOutputs | 0x0001;    // set the
first onboard-output
system.dwOutputs := system.dwOutputs & (~0x0001); // clear the
first onboard-output
system.dwOutputs := system.dwOutputs EXOR 0x0001; // toggle
the first onboard-output
```

A sample script which uses digital onboard I/O is the 'TrafficLight' application .

`system.dwFirmware`

Retrieves the *hardware-specific* firmware '**article number**' as a 32-bit integer value. Example:

```
  print("FW-Art-Nr.=",system.dwFirmware);
```

Output (when the above example is executed on different target systems):

```
FW-Art-Nr.=11314        (on an MKT-View II with 'CANdb' firmware)
FW-Art-Nr.=11315        (on an MKT-View II with 'CANopen' firmware)
FW-Art-Nr.=11392        (on an MKT-View III with 'CANdb' firmware)
FW-Art-Nr.=11393        (on an MKT-View III with 'CANopen' firmware)
FW-Art-Nr.=11222        (when running on a PC in the 'simulator')
```

`system.dwVersion`

Retrieves the firmware **version number** (in the target device) as a 32-bit integer value. Format:

```
  bits 31 .. 24 = major version number (Hauptversionsnummer)
```

```
    bits 23 .. 16 = minor version number (Nebenversionsnummer)
    bits 15 ..  8 = revision number
    bits  7 ..  0 = build nummer
```

Example:

```
  print("FW-Version=0x"+hex(system.dwVersion,8));
```

-> output: `FW-Version=0x01020304` (if the firmware version was "V1.2.3 - build 4")

`system.serial_nr`

Retrieves the device's serial number as an integer value.
If a device doesn't support unique serial numbers (stored in an EEPROM), the result is zero.
Example:

```
  print("Serial Number="+itoa(system.serial_nr,5));
```

`system.nv[ 0..31 ]`

Accesses one of the 32 'non-volatile values', like the nv-function in the UPT display interpreter. The same restrictions concerning EEPROM cell endurance apply. Details here.
Setting a new value as in the following example doesn't immediately 'write' the value into the system's configuration EEPROM, but into a latch:

```
   system.nv[0] := 12345; // write 32-bit integer into the non-volatile memory latch
```

To store the contents of the 32 'latches' (integer values) permanently in the EEPROM, and you are really sure that no other `nv[]`-values need to be set in future, carefully invoke the following command to write the contents of he latches into the sluggish EEPROM (which may take a considerable amount of time, depending on temperature and 'fitness' of the EEPROM chip):

```
   system.nv_save; // write all modified nv locations into the EEPROM
```

`system.resources`

This function is just an aid for debugging. It returns a combined indicator of 'remaining system resources', measured in **percent**.
The value is the minimum of the following script-related system parameters:

- Remaining stack space (used for local script variables, etc)
- Remaining dynamic memory ("heap") available for the script

If the system resources drop below 10 percent, the script may contain a bug (for example, illegal recursion, allocate too many strings or arrays, etc).
The reason can be examined in the debugger/simulator (memory usage display), integrated in the programming tool.
The function system.resources works the same way in the 'real' target as well as in the programming tool.

`system.timestamp`

Retrieves the system's local timestamp generator value. The same generator also produces the timestamps for the CAN driver, thus by comparing system.timestamp with the timestamp in a received CAN message, you can tell, down to the fraction of a millisecond, how much time has elapsed since the reception of that message. Together with the wait_ms() command, you can also use this function to synchronize the activity of the script.
Example: Send a precisely timed 'answer' for a CAN message :

```
display.pause := TRUE;  // don't let the display-interpreter
interfere for a short time
Tdiff := can_rx_msg.tim - system.timestamp; // timestamp
difference between 'now'
                                            // and a certain
CAN message reception
Tdiff := (1000*Tdiff)/cTimestampFrequency; // convert to
milliseconds
wait_ms(100-Tdiff);    // wait until 100 ms have passed since
CAN msg reception
can_transmit;
display.pause := FALSE; // resume normal display operation
```

Note: This example isn't 100 percent bullet-proof. It only works if this code is executed within 100 milliseconds after the time of a CAN message reception. To mininize the risk of wasting too much time for the display-update, the display should be paused immediately after reception of the CAN message which shall be 'answered'. Remember that the display terminal isn't a programmable logic controller with 'guaranteed' maximum latencies, even if there is a fast pre-emptive multitasking kernel running "under the hood".

The 32-bit timestamp is generated by a hardware timer, which starts at zero during power-on, and then increments at a hardware-depending frequency specified as constant `cTimestampFrequency`. The timer frequency is typically in the range of 40 kHz, so a signed 32-bit number will overflow from 0x7FFFFFFF (large positive value) to 0x80000000 (very negative value) after about 2^31 / 40000 Hz = 53687 seconds, or 14 hours. Despite that, the 32-bit integer arithmetic (as in the example shown above) will still give a valid DIFFERENCE between two timestamps, even if a timestamp wrapped from 0x7FFFFFFF to 0x80000000 or from 0xFFFFFFFF to 0x00000000 . This is the reason why you *must not* convert a timestamp into seconds (or any other unit) *before* calculating a timestamp-difference. First calculate the difference (as in the example above, "Tdiff := can_rx_msg.time - system.timestamp"), then convert the timestamp difference into any unit you like (as in the example, "Tdiff := (1000*Tdiff) / cTimestampFrequency") .

system.unix_time

If the system is equipped with a battery-buffered real time clock (RTC), this function returns the system's current date and time in 'Unix Time' format.
The 'Unix Time' aka 'Unix Second' is defined as

the number of seconds since midnight January 1st, 1970 **(1970-01-01 00:00:00) .**

Beware of the "Unix Millenium Bug" (Year 2038 Bug) which will affect any system which (as many of today's Unix / Linux systems) use a signed 32-bit integer to store the Unix Time ! For details, see the TimeTest.cvt example .
The 'system' (terminal) doesn't care about timezones, so we suggest you let the built-in real-

time clock run in UTC (universal time). Only in that case, system.unix_time (and system.unix_time_boot, see below) can really return date and time in UTC, as it should. See also: time.unix_to_str, Date and time conversions, Modified Julian Date (MJD)

`system.unix_time_boot`

Returns the system's date and time in 'Unix Time' format, at the time the system (programmable display) was booted, and when the timestamp generator (system.timestamp) was zero .

`system.audio_ptt`

Controls a relais for the audio output's 'Push-To-Talk'-feature. Only exists in a few 'special' devices.

`system.feed_watchdog`

This command should only be used *if really necessary*, for example if an event handler, or a function invoked via backslash sequence in a display element's format string, is expected to require ***more than 200 milliseconds for execution***. Fortunately, in most applications this is hardly ever necessary, because any 'long-lasting' operations can easily be moved into the script's main loop (main task), and in the event handler, you will only set a flag (variable) which can be polled in the main loop. Setting a variable takes much less than a millisecond, so you don't need to feed the watchdog in a well-designed event handler.
In the following paragraph, the term 'event handler' also applies to a function invoked via backslash sequence in the format string of a display element (as in the 'GetText' example).
Technical details follow...
Endless loops, and long calulations must be avoided in event handlers, because they will render the device inoperable (or, from an operator's point of view 'it crashes'). To avoid this (in case of an erroneous script), the run time system will terminate the function call, if the function (event handler) doesn't return to the caller fast enough (i.e. in less than 200 milliseconds). In the normal script context, there is no such limit because the pseudo-multitasking guarantees that the device stays 'responsive' even if the script gets stuck in an endless loop without calling any 'waiting' function.
If the maximum event handler execution time is not sufficient, the forced termination can be avoided with a command like this:

system.feed_watchdog(500); // Feed the script's watchdog for another 500 milliseconds

As already mentioned, this consequence of doing this may be a sluggish user response, but also protocol timeouts (because as long as the event handler is busy, the device will not do much else). So *whereever possible, avoid this command*, and re-write your event handlers so they return to the caller as fast as possible. Then you won't need to feed the watchdog at all.

`getkey`

Reads the next key from the UPT's keyboard buffer. The same buffer is also used by the display interpreter's kc function (!), so reading a key through getkey also removes it for the 'kc'-function, and vice versa !
If the keyboard buffer was not empty, getkey will return a non-zero value. Usually, this value is one of the *key*-constants which you can use in a select-case list to implement "handlers" for

the individual keys. Note that not all keyboards support all possible keys ... some of MKT's keyboards only have function keys (keyF1 to keyF3), others have only cursor keys (keyUp, keyLeft, keyRight and keyDown) and / or keyEnter and keyEscape (Enter alias "Return", sometimes this key is generated by pressing the rotary button).
For numeric keyboards, use key0 to key9 .
Don't make any assumption about the actual key values, they may be hardware-specific !
Only use the *key*-constants ! The only value returned by getkey which will definitely never change in future is 0 (zero), which means "no key has been pressed since the last call" .
An example for the getkey function, used in a select-case list, can be found in the test application TScreenTest.cvt , and in the 'QuadBlocks' demo (to control the game via keyboard).
For more advanced control (for example, to detect when a key has been pressed and released), use the low-level event handlers OnKeyDown and OnKeyUp instead.

See also : display (functions), keywords, contents .

9.6 3.10.7 Date and Time conversions

The following built-in functions and procedures can be used for basic date- and time conversions aka "calendar" functions :

`time.unix_to_str(string format, int unix_date_and_time)`

This *function* converts a date (precisely, date and time in Unix seconds) into a string. The format is specified by means of a format string (first argument), like:
"YYYY-MM-DD hh:mm:ss" : produces an ISO 8601-compliant representation with date *and* time.
(without a 'T' between date and time because the 'T' is ugly ... see next example)
"YYYY-MM-DDThh:mm:ss" : would be fully ISO 8601-compliant, but looks ugly
"DD.MM.YYYY" : produces an 'unlogic' date format which is unfortunately common in Germany.
"MM-DD-YYYY" : another 'unlogic' but unfortunately common date format.
"MMM-DD-YYYY" : similar but less ambiguous: Three letters (not digits) for the month.
(MMM, all in upper case, expands to JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)
"Mmm DD, YYYY" : similar as the traditional american date format, but only three letters for the month.
(Mmm, mixed upper/lower case, expands to Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec)

We suggest to use ISO-compatible date- and time formats only; especially if you are a German company with customers on the other side of the pond.
What do you, and what would your customer make of 3/12/2001 ? The 12th day of March, or the 3rd day of December ?

Example:

```
    print("It is now ", time.unix_to_str("YYYY-MM-DD
hh:mm:ss",system.unix_time) );
```

`time.date_to_mjd(in int year, in int month, in int day)`

This *function* converts (combines) a date consisting of year (1858..2113), month (1..12), and day-of-month (1..31) into the Modified Julian Date (MJD, defined futher below). The function result ("return value") is the MJD.

`time.mjd_to_date(in int mjd, out int year, out int month, out int day)`

This *procedure* converts (splits) a Modified Julian Date number (MJD) into a normal Gregorian date, consisting of year (1858..2113), month (1..12), and day-of-month (1..31) . All outputs are specified as 'outputs' in the argument list, there is no 'function result' aka 'return value'.

The Modified Julian Date (MJD) is commonly defined as

the number of days since midnight November 17, anno 1858 **(1858-11-17 00:00:00 in ISO 8601 format) .**

It is widely used to calculate differences between dates, becauses the difference between two MJDs is the number of days (!) between their calendar dates.
The MJD can easily be converted into a 'Unix' time, because the UNIX BIRTHDATE (1970-01-01) as MJD (day) is 40587 . In contrast to MJD, the 'Unix time' counts the number of seconds since that birthdate, so the conversion from MJD to 'Unix seconds' is straightforward ... see example programs/script_demos/TimeTest.cvt .

9.7 3.10.8 Commands to control the Trace History

The following procedures and functions can be used to control the Trace History from the script. Their main intention is for debugging, development, and to track CAN bus problems.

`trace.print( <Parameter> )`

Prints the specified parameters (strings and numeric values) as a single line of text into the device's Trace History.
This is possible in the simulator (programming tool) as well as on a 'real' target, but the target must have a 32-bit-CPU (ARM), and the firmware must be from July 2012 or later.
Example (prints the current date and time into the Trace History):

```
  trace.print( "Date,Time: ", time.unix_to_str( "YYYY-MM-DD
hh:mm:ss", system.unix_time ) );
```

`trace.enable`

With this formal variable ("flags"), the script can stop or resume the trace history, for example to prevent CAN messages being appended after the script (application) found out that 'something went wrong' already, and all further CAN traffic is not relevant to track down the cause.
The trace-enable-flags (trace.enable) is actually a bitwise-OR combination of the following constants:

- traceCAN1 : add CAN messages, 1st bus, to the trace history
- traceCAN2 : add CAN messages, 2nd bus, to the trace history
- traceCAN_UDP: add CAN-via-UDP messages to the trace history
- ~~traceFlexRay: add Flexray-via-UDP messages to the trace history~~
- traceUDP : add generic UDP/IP frames to the trace history
- traceTCP : add generic TCP/IP frames to the trace history

With trace.enable := 0 (zero), none of the above events will be added to the trace history (i.e. trace stopped, only trace.print will be able to append more items to the trace).

Example (script code):

```
  trace.enable := traceCAN1 + traceCAN2; // trace messages
from CAN1 and CAN2
  if( can_rx_fifo_usage > 500 ) then
    trace.print( "CAN FIFO usage: ", can_rx_fifo_usage );
    trace.enable := 0; // don't add more CAN messages to the
trace history
  endif;
```

By default (after power-on) the trace history is **enabled** for CAN1 and CAN2, which means all CAN messages registered for reception, and all CAN messages transmitted by the device are appended to the history.
Setting **trace.enable=0** does *not* affect the trace.print command; the script can always 'print' into the Trace History via command.

trace.stop_when_full

If this flag is set (by the script), the trace history will be automatically stopped as soon as the trace buffer is (almost) full.
This feature can be used to catch the initial part of a CAN conversation. An example is in the application 'TraceTest.cvt':

```
// Select the items which shall be displayed in the trace history:
trace.enable := traceCAN1 + traceCAN2; // trace messages from CAN1 and CAN2

// Let the TRACE HISTORY stop when the trace-buffer is full
//  (instead of overwriting the oldest entries) :
trace.stop_when_full := TRUE;
```

Note: With the option 'trace.stop_when_full := TRUE', the trace-history will be stopped internally by setting trace.enable := 0 (in the firmware).
To resume acquisition, set trace.enable as shown in the example above.
By default (after power-on), trace.stop_when_full is FALSE.

trace.num_entries

Returns the *current* number of entries in the trace history.
As long as the trace history is enabled, this value may increase up to (almost!) trace.max_entries.

trace.max_entries

Returns the *maximum* number of entries in the trace history.
This value is constant (for a certain device), but it may depend on the device firmware due to memory constraints.
The example in 'TraceTest.cvt' (details below) uses this value as argument for a modulo-operation, to limit the circular buffer indices:

```
iTailIndex := (iTailIndex+1) % trace.max_entries;
```

trace.oldest_index

Returns the trace buffer index where the OLDEST entry has been stored.
Due to the 'circular' nature of the buffer, the oldest entry isn't necessarily at index zero !

trace.head_index

Returns the trace buffer index where the NEXT (newest) entry **will** (future!) be stored.
It also marks the *endstop* when listing the trace buffer in the script itself.
Example (from the application 'TraceTest.cvt'):

```
iTailIndex := trace.oldest_index; // start listing trace-entries HERE
while( iTailIndex != trace.head_index )
   print( trace.entry[iTailIndex], "\r\n" );  // dump next entry to screen
   iTailIndex := (iTailIndex+1) % trace.max_entries; // increment "tail"
index
   // (iTailIndex wraps from 'max_entries minus one' to zero,
   //  because the trace buffer is organized like a CIRULAR ARRAY )
endwhile;
```

Note: If **trace.head_index** is equal to **trace.oldest_index**, the trace history is *empty*.

trace.entry[n]

Retrieves the n-th entry in the trace history buffer as a string.
The oldest entry is at index n=trace.oldest_index .
A 'headline', compatible with the display format, can be retrieved by trace.entry[-1].
A 'separator line', consisting of a string of dashes, can be retrieved by trace.entry[-2].

```
print( trace.entry[-1], "\r\n" ); // print a 'headline' for the trace
display
print( trace.entry[-2], "\r\n" ); // print a 'separator' for the trace
display
```

For a complete example, see trace.head_index, where this function is used to dump the trace-history to a text panel.

trace.can_blacklist[i]

Retrieves the n-th entry in the blacklist of CAN-IDs, which can exclude up to 10 individual CAN message identifiers from the trace history (display).
At the time of this writing (2013-11-27), the index (i) may be 0 to 9, because the blacklist is limited to a maximum of **ten** entries.

An example which exclude certain CAN message identifiers from the trace history via script is in the test/demo application TraceTest.CVT.

`trace.file_index`

Gets or sets the file-sequence-number for saving the trace history as a text file.
To save the trace history as text file, you can use the command trace.save_as_file mentioned below.

`trace.save_as_file`

This command saves the contents of the trace history buffer as a text file on the memory card.
The same can be achived 'manually' (by the operator) as explained here.
With each new file saved, the file-sequence-number (exposed as trace.file_index) is incremented by one.
Only certain devices support this feature !

`trace.clear`

Clears (erases) the trace history buffer.

9.8 3.10.9 Interaction between Script and Internet Protocol Stack

Most devices with an Ethernet port also have an integrated Internet protocol stack (with TCP/IP). The functions and event handlers presented in this chapter can be used for a 'direct' interaction between the script (application) and the IP stack.
For the normal use (TCP/IP used for the embedded web server) it's not necessary to have special commands in the script for controlling the TCP/IP stack. For example, files uploaded into the RAMDISK via web server (HTTP-POST) can be processed by the script using the standard file I/O-functions; and files which were written into the RAMDISK by the user's script can be read via web server (HTTP-GET) from the device.

9.8.1 3.10.9.1 Overview of Internet Socket API functions

In addition to the web server (which doesn't depend on the script language at all), a script application can implement its own 'IP based' functionality. For this purpose, the script language contains a small subset of the Berkeley Socket API. The following list is just an overwiev of the most important socket-based functions. For details about Berkeley Sockets, consult other literature, or study the examples further below.

iSock := inet.socket(int address_family,int type,int protocol);

Creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it.
The returned value, traditionally called a 'socket' in resemblance to a telephone socket, is an integer value which identifies the communication endpoint. It must be stored in an integer variable until the socket is closed (i.e. "unplugged") again.
Most functions listed below expect the socket number as their first input argument.

inet.close(int socket)

Causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.

Unlike the other socket API functions, inet.close does not return a value.
If certain network operations are still outstanding, a closed socket may not be available immediately for other tasks, i.e. given back to the pool of 'free' sockets. For example, a socket which was successfully CONNECTED to a remote peer will first enter the CLOSING state, before being 'really' CLOSED.

result := inet.bind(int socket, int port_number)

Used on the *server* side, and associates a socket with the specified local port number. In contrast to the Berkeley socket API, inet.bind() doesn't care for IP addresses because the server's IP address is always the same as the device's IP address.

result := inet.listen(int socket, int nConnections)

Used on the *server* side, and causes a bound TCP socket to enter listening state.

iAcceptedSocket := inet.accept(int iListeningSocket)

Used on the *server* side. Accepts an 'incoming call' from a remote client, and creates a new socket associated with the socket address pair of this connection.
The returned value is a new socket, which should later be closed/freed (inet.close) to prevent running out of system resources.

his_name := inet.getpeername(int iAcceptedSocket)

Typically used on the *server* side, after successfully accepting a connection.
This function shall retrieve the peer name ("IP address) of the specified socket *as a string*.
Notes: The Berkeley API uses a fancy structure to store the result for this function; but here the result is a simple string.
In this context, a peer is 'the guy at the other end of the line', i.e. the remote client.

iSockState := inet.getsockstate(int socket)

Retrieves the current state of the specified socket.
The result may be one of the following symbolic script constants 'SCKS_...' - see socket states.

result := inet.connect(int socket, int timeout_ms, string destination)

Used on the *client* side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.
This function usually requires several hundred milliseconds, depending on the network and the protocol, because often the TCP/IP protocol stack must resolve the remote server's IP address, or name (using ARP and/or DNS). The second parameter (timeout_ms) specifies the maximum number of milliseconds after which inet.connect must return (with or without success).
The return value indicates success (0=SOCK_SUCCESS), or a negative error code defined here.

result := inet.send(int socket, int timeout_ms, input_arguments)

Sends data to a remote socket. If the network transmit buffer is full, the command may block the caller up to <timeout_ms> milliseconds. If the connection bandwidth it large enough, *inet.send* will not block at all because the actual transmission of data takes place in the background (in a different task).

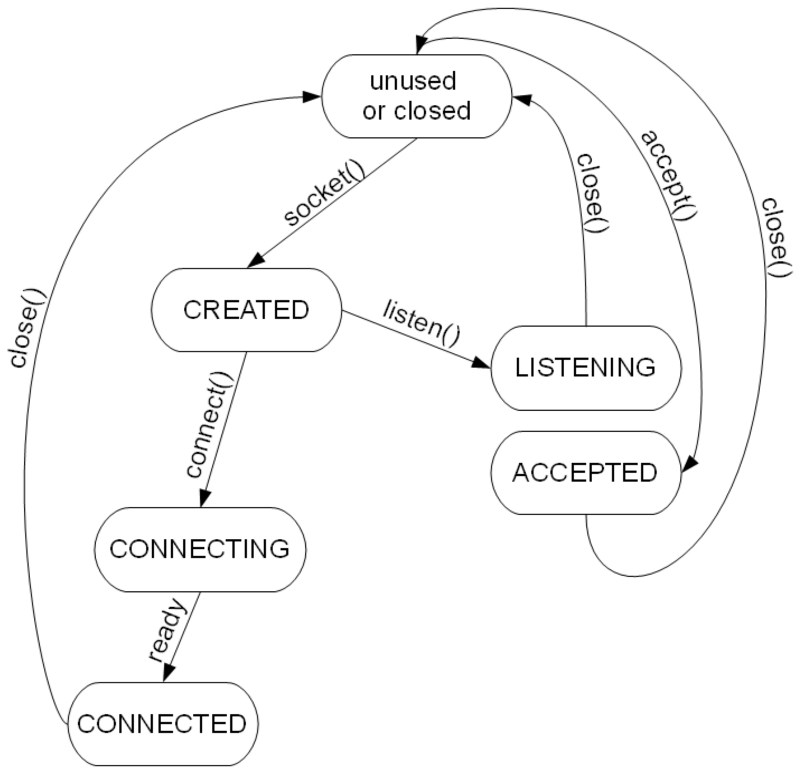
result := inet.recv(int socket, int timeout_ms, output_arguments)

Receives data from a remote socket. Returns the number of bytes received (if any); or a negative error code (one of the SOCK_ERROR constants). If the network receive buffer is empty, the command may block the caller for up to <timeout_ms> milliseconds to wait for the reception of data. The output arguments must be *passed by reference*, using the address-taking operator. Details about receiving data via 'inet.recv' are here.

Examples for the socket style API can be found in the following subchapters, and in the application script_demos/InetDemo.cvt, which contains a small, socket-based TCP client and server, written entirely in the script language.
Details on some of the 'inet' functions follow in the next chapters.

9.8.2 3.10.9.2 Internet socket state diagram

The following diagram shows the basic states of a socket, and their transistions.
Not shown here for clarity: Transitions into the (unrecoverable) error state.



The current state of a socket can be retrieved with via inet.getsockstate(<socket>).
The result (an integer value) is one of the following symbolic constants in the script language:

SCKS_NOT_IN_USE
: this socket is currently not in use.

SCKS_CREATED
: socket has been created but neither listening (server side) nor connecting (client side) yet

SCKS_LISTENING
: server side: called 'inet.listen' but not 'inet.accept' yet

SCKS_ACCEPTED
: server side: called 'inet.accept', with success (!). Note that accept() allocates a new socket from a pool,
thus the 'accepted' socket is not the same as the 'listening' socket, and there is no transition from LISTENING to ACCEPTED for the listening socket !

SCKS_CONNECTING
: client side: trying to connect to a remote server

SCKS_CONNECTED
: client side: successfully connected to a remote server

SCKS_CLOSING
: either side: closing the socket, but some network operations may be still pending

SCKS_CLOSED
: either side: the connection is definitely closed by someone

SCKS_ERROR
: either side: an (unexpected) error occurred, connection broke down, etc.
The application should close the socket, and if necessary try to reconnect.

9.8.3 3.10.9.3 Error codes for the Internet Socket Services

The following internet-socket related err codes are available as symbolic constants in the script language.
Their values are not compatible with error codes specified in the Berkeley socket services, so don't make any assumption about the actual values (except that SOCK_SUCCESS is ZERO, and all other error codes have *negative* values), and use *only* these symbolic constants in your code:

SOCK_SUCCESS
: Success, or operation completed. Since this 'error code' is not an error at all, its value is zero.

SOCK_ERROR
: General Error

SOCK_EINVALID
: Invalid socket descriptor

SOCK_EINVALIDPARA
: Invalid parameter

SOCK_EWOULDBLOCK
: Caller would have been blocked (if it was a 'blocking' socket, i.e. completion is pending)

SOCK_EMEMNOTAVAIL
: Not enough memory in memory pool, or too many handles or 'sockets' in use

SOCK_ECLOSED
Connection is closed or aborted

SOCK_ELOCKED
Socket is locked in RTX environment

SOCK_ETIMEOUT
Timeout (on a socket, during address resolution, name lookup, connection set-up, or whatever)

SOCK_EINPROGRESS
Host Name resolving in progress

SOCK_ENONAME
Host Name not existing

No connection could be made because the target machine actively refused it

SOCK_ENOTSUPPORTED
a particular function, or a combination of options, is not supported (/ yet ?)

Note: A function to convert these error codes into human-readable text is contained in the 'Internet Demo' application (script_demos/InetDemo.cvt) .

The following chapters contain details about some of the internet related functions in the script language.

9.8.4 3.10.9.4 inet.socket(int address_family, int socket_type, int protocol)

This function creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it.

The returned value, traditionally called a 'socket' in resemblance to a telephone socket, is an integer value which identifies the communication endpoint. It must be stored in an integer variable until the socket is closed (i.e. "unplugged") again. Negative return values indicate an error (see error codes listed here).

Parameters:

address_family : One of the AF_ constants, similar to the Berkeley socket API.
AF_INET = Internet Protocol (V4). This is currently the only supported address family.

socket_type : One of the SOCK_ constants, similar to the Berkeley socket API.
SOCK_STREAM = Stream socket (Connection oriented, for example TCP)
SOCK_DGRAM = Datagram Socket (Connectionless, for example UDP)

protocol : One of the IPPROTO_ constants, similar to the Berkeley socket API.

IPPROTO_TCP = TCP/IP (used together with socket_type SOCK_STREAM)
IPPROTO_UDP = UDP/IP (used together with socket_type SOCK_DGRAM)

Note: Any other combination of 'address family', 'socket type', and 'protocol' beside those listed above is expected NOT to work properly !

Example:

```
  iListeningSocket := inet.socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
  if ( iListeningSocket < 0 ) then    // negative result means ERROR !
     print("Could not create a socket !");
  endif;
```

9.8.5 3.10.9.5 inet.connect(int socket, int timeout_ms, string destination)

Used on the *client* side. Assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.

This function usually requires several hundred milliseconds, depending on the network and the protocol, because often the TCP/IP protocol stack must resolve the remote server's IP address, or name (using ARP and/or DNS). The second parameter (timeout_ms) specifies the maximum number of milliseconds after which inet.connect must return (with or without success).
The return value indicates success (0=SOCK_SUCCESS), or a negative error code defined here.

9.8.6 3.10.9.6 inet.send(int socket, int timeout_ms, input_arguments)

Used both on the *client*- and *server*- side.
Sends data to a remote socket ('peer').
If the network transmit buffer is full, the command may block the caller up to <timeout_ms> milliseconds. Regardless of being successful or not, inet.send() will always return after that interval (or earlier). If the connection bandwidth it large enough, *inet.send* will not block at all because the actual transmission of data takes place in the background (in a different task).
The return value indicates success (0=SOCK_SUCCESS), or a negative error code defined here.

9.8.7 3.10.9.7 inet.recv(int socket, int timeout_ms, output_arguments)

Usable on both the *client*- and *server*- side.
Receives data from a remote socket.
Return value: The number of bytes received (if any); or a negative error code (one of the SOCK_ERROR constants).
If the network receive buffer is empty, the command may block the caller for up to <timeout_ms> milliseconds to wait for the reception of data.
The output arguments must be *passed by reference* (not, as usual, *passed by value*). To achieve this, use the address-taking operator (&) as prefix before the names of the destination variable(s).
Example (from 'InetDemo.cvt') :

```
    proc RunTcpClient   // minimalistic 'TCP client' .
       local int iResult;
       local string s;
        (...)
       iResult := inet.recv( iClientSocket, 20/*ms*/, &s );
       if( iResult > 0) then // received something -> process it
           print( s );
       endif;
    endproc;
```

Regardless of being successful or not, inet.recv() will always return after that interval (or earlier).

In most higher internet protocol layers (HTTP, FTP, ..), lines of text are exchanged between client and server. Thus the output arguments will almost exclusively be *text strings*. Each line of text must end with <CR><LF> (Carriage Return followed by 'Linefeed' aka New Line, hexadecimal 0x0D 0x0A).
Thus, the default 'separator' when receiving strings from a socket using inet.recv is this end-of-line marker. Unfortunately, some applications stubbornly ignore this. To simplify the treatment of different END-OF-LINE markers, inet.recv() uses the following end-of-string markers:

- A zero-byte is always an end-of-string marker, as in the "C" programming language.
- <CR> (carriage return) immediately followed <LF> also marks the end of a string (on reception), and both of these characters are appended to the end of the string.
- <CR> *not* followed by <LF> also marks the end of a string (on reception), and is appended (as a single character) to the end of the string.
- <LF> ('linefeed', 0x0A) without a preceding <CR> is treated like a normal ASCII character, and does *not* mark the end of a string;
  unless you explicitly define this character as separator after creating the socket.

9.8.8 3.10.9.8 JSON (Javascript Object Notation)

Even though the script language is in no way compatible with Javascript, it will support JSON (planned for the end of 2013, or maybe early 2014).
This will, for example, allow a seamless implementation of an interface to openABK ('offenes Anzeige- und Bedien-Konzept'), initiated by BMW.
Future plan : With sufficient demand from other users, there will be commands in the script language to communicate via openABK 'directly'.

9.9 3.10.10 Interaction between Script and the CANopen Protocol Stack

Note: The functions described below are only available since August 2013 in devices with built-in CANopen protocol stack, and in the 'UPT Programming Tool II' (as simulator) !

The following commands and functions with the prefix 'cop.' (short for 'CANopen') are implemented in the script language:

cop.obd(Index,Subindex[,Data_type])

Accesses an object in the CANopen device's *own* (local) object dictionary (OD).
Since no network operations are involved, this function returns to the caller immediately.

Because in a CANopen device *almost anything* can be controlled via the OD, the script can use this command (as a formal assignment, i.e. write-access) to modify *its own* behaviour regarding its CANopen communication. Just a few examples:

- Reconfigure the process data communucation via PDO-**Communication**-Parameter (Object indices 0x1400=RPDO1 CommPar, 0x1401=RPDO2, .. , 0x1800=TPDO1, 0x1801=TPDO2, ..)
- Reprogram the *contents* of the process data telegrams by virtue of the PDO-**Mapping-**Parameter
  (Object indices 0x1600=RPDO1 Mapping, 0x1601=RPDO2, .. , 0x1A00=TPDO1, 0x1A01=TPDO2, ..)
- Reconfigure the SDO-Clients (used to communicate via cop.sdo)
  (Object indices 0x1280=first SDO-Client, 0x1281=second SDO-Client, etc..)
- Reconfigure the SDO-Server (which allow *other devices* to access the terminal's OD)
  (Object indices 0x1200=first SDO-Server, 0x1201=second SDO-Server, etc..)

An example using cop.obd to modify the PDO mapping is in CANopen1.upt .

cop.sdo(Index,Subindex[,Data_type][,Timeout_in_milliseconds]) ,
cop.sdo2(SDO-Channel,Index,Subindex[,Data_type][,Timeout_in_milliseconds])

Accesses an object in a *remote* CANopen device's object dictionary via SDO ('**S**ervice **D**ata **O**bject').
The simple variant (cop.sdo) always uses the *first* SDO client, cop.sdo2 can use any of the SDO clients (as far as supported in the firmware) identified by the zero-based SDO-channel number.
Because network operations are involved, these functions will take some time to complete. During this time, the caller (script) will be blocked for several milliseconds, and the script will be switched into the 'waiting' state. The waiting time may be interrupted by event handlers. For this reason, **cop.sdo must not be used in event handlers** itself.
Use cop.sdo only in the script's main task (main loop) !

The SDO-clients (and thus 'cop.sdo' in your scripts) support read- and write-access.
For read-access, use *cop.sdo* on the right side of an assignment-operator ("LVALUE"), for write-access, use it on the left side ("RVALUE") as in the following examples:

```
   // Object 0x1018, subindex 0x01 = "Vendor ID", part of the "Identity
Object", see CiA 301 :
   my_vendor_id  := cop.obd( 0x1018, 0x01, dtDWord );  // read vendor ID
from this device's own OD
   his_vendor_id := cop.sdo( 0x1018, 0x01, dtDWord );  // read vendor ID
from a REMOTE device's OD
   cop.error_code := 0;  // Clear old 'first' error code ('abort code')
before the next SDO access
   cop.sdo(od_index, subindex ) := iWriteValue;        // try to write into
remote object via SDO
   if( cop.error_code <> 0 ) then  // if the previous SDO accesses were ok,
cop.error_code is zero
      print("\r\nSDO access error, abort code =
0x",hex(cop.error_code,8) ); // show abort code (hex)
   endif;
   iReadBackValue := cop.sdo(od_index, subindex, dtInteger ); // try to
read from a remote object via SDO
   if ( iWriteValue <> iReadBackValue ) then
      print("\r\nSDO error: Read-back value (",iReadBackValue,") is
different from written value (",iWriteValue,") !");
   endif;
   print("\r\n My name is ", cop.obd( 0x1008,0, dtString ) );  // "Michael
Caine" ? No, but..
   print("\r\n His name is ",cop.sdo( 0x1008,0, dtString ) );  // ..the
'Manufacturer Device Name'
```

A complete example using cop.sdo is in CANopen1.upt .
SDO clients are usually configured in the programming tool as explained here.
The various CANopen SDO Protocols are specified in CANopen CiA 301.

cop.error_code

This variable will be set when an SDO protocol error ('abort code') is indicated (locally or via CAN from a remote server). The error code is usually displayed as an 8-digit hexadecimal value (see excerpt from CiA 301 below), or can be translated into a human-readable string with a select-case list as in the demo CANopen1.upt ("ErrorCodeToString").
If an SDO transfer is completed *without* an error, the value stored in cop.error_code does *not* change. As long as cop.error_code is nonzero, the value will also not change (even if a subsequent error occurrs).
Together with **cop.error_code**, the variables **cop.error_index** (= CANopen OD index of the object which caused the error) and **cop.error_subindex** (= subindex of the object which caused the error) will be updated.
To clear (or acknowledge) the error in the script, set cop.error_code to zero as in the following example:

```
   cop.error_code := 0;  // Clear old CANopen error code (usually an SDO
abort code; here: 0 = "no error")
```

The CANopen-SDO-Abort-Codes are specified in CANopen CiA 301. Here's a *small excerpt* :

Abort Code	Description

0x05030000	Toggle bit not alternated
0x05040000	SDO protocol timed out
0x06010000	Unsupported access to an object
0x06010001	Attempted to read a write-only object
0x06010002	Attempted to write a read-only object
0x06020000	Object does not exist in the object dictionary
0x08000000	General error

cop.nmt_state

Retrieves the current NMT state of the CANopen node (built inside the programmable device).
The return value may be one of the following constants:

- **cNmtStateBootup** (0) :
  The CANopen device is initialising itself; it cannot communicate, and the object dictionary doesn't exist yet.
- **cNmtStatePreOperational** (127) :
  In the NMT state Pre-operational, communication via SDOs is possible. PDOs do not exist, so PDO communication is not allowed. (...)
  The CANopen device may be switched into the NMT state Operational directly by sending the NMT service start remote node or by means of local control.
- **cNmtStateOperational** (5) :
  All "communication objects" (CANopen-slang) are active. Transitioning to the NMT state Operational creates all PDOs; the "constructor" uses the parameters as described in the object dictionary.
- **cNmtStateStopped** (4) :
  The CANopen device is forced to stop the communication altogether (except node guarding and heartbeat, if active)

Details about the 'NMT state machine' of a CANopen device could be found in CiA 301 (formerly known as 'DS 301'..), Version 4.2.0 (Februrary 2011), chapter 7.3.2, pages 83 to 85. The restrictive terms of use (in CiA 301) don't allow us to duplicate that information here; so please obtain a copy of that document from CiA yourself.

cop.node_id

Retrieves the CANopen node-ID (1..127) of the device on which the script is running.
The value is read-only. To modify a device's node-ID, use the system setup.

cop.SendNMTCommand(int node_id, int wanted_nmt_state)

Sends an NMT message *to the CANopen network* to switch the desired node(s) into the wanted NMT state.
Valid CANopen nodes IDs are 1 to 127. In addition, for the NMT (Network Management) protocol, node-ID zero can be used to address 'all nodes' in the network.

If the node-ID matches the local node ID, the local node also transits to the new state.
The NMT state can be one of the following constants (which are also used for cop.nmt_state):

- **cNmtStateBootup** : force re-booting (Bootup), actually sends the 'Reset Node' command to one or all slaves.
- **cNmtStatePreOperational** : switch one or all slaves into the 'Pre-Operational' state.
- **cNmtStateOperational** : switch one or all slaves into the 'Operational' state.
- **cNmtStateStopped** : switch one or all slaves into the 'Stopped' state.

Return value (if cop.SendNMTCommand was called as a *function*) :
TRUE = ok
FALSE= error (function not available, illegal node ID, illegal NMT command, ...)

See also: cop.nmt_state : Retrieves the device's own momentary NMT state.

cop.SetPDOEvent(int pdo_comm_par_index)

Sets an 'event-'flag for a certain PDO which may cause immediate transmission.
The PDO is identified by the CANopen OD-index of its 'PDO communication parameter', for example:
0x1800 = first transmit-PDO, **0x1801** = second transmit-PDO, etc.
If setting a PDO's 'event' really causes an immediate transmission depends on the PDO's *transmission type*. The transmission type is usually defined in the programming tool's PDO-communication-parameter dialog. It may be also affected by a TPDO's optional *inhibit time* (which limits the maximum frequency at which a PDO can be transmitted).
Return value (if cop.SetPDOEvent was called as a *function*) :
TRUE = ok (PDO-event-flag was successfully SET)
FALSE= error (function not available, illegal CANopen-OD-Index, etc...)

See also:

- The display terminal's own (local) CANopen object dictionary (OD)
- Object 0x5001 in the CANopen OD : PDO-mappable 'Keyboard Matrix Bits'
- Features of programmable terminals with "CANopen V4"
- PDO-Mapping (Defining the contents of 'Process Data Objects')
- SDO Abort Codes (Error codes used when communicating via CANopen SDO)
- CANopen specifications from CiA (CAN in Automation), most important: CiA 301

9.10 3.10.11 Extensions of the script language for J1939

The support for SAE J1939 *in the script language* isn't finished yet. Until then, some preliminary information about how to implement parts of the J1939 protocol can only be found in the documentation in German language .

10 3.11 Event Handling (handling system messages and similar events *in the script*)

As a replacement for the 'event definitions' in the display interpreter, the script language can be used to react when the user 'does something' on a display page, press a key, operate the touchscreen, or the rotary encoder. The script may even intercept(!) certain events, i.e. disable the default message handler for that event.

In the script language, system messages / events can be handled by simply adding your own message handler. The return value of a message handler (function) tells the system if the message shall be discarded (because your script has processed it, and doesn't want the message be passed to the 'default' message handler). More on this later. Let's begin with the "lowest level" of message processing: Keyboard events, rotary encoder events, and (depending on the device capabilities) touchscreen events.

Note:

Message handlers in the script language will **interrupt** the normal program flow for a few milliseconds.
An event handler must return as soon as possible - ideally after less than 50 milliseconds.
If the script gets stuck in an event handler, the handler will be terminated ("killed") after approximately 500 ms, and the event will be handled by the system instead.
So keep your message handlers as short as possible, and return as quickly as possible !

To avoid 'slow processing' in an event handler, just set a signal ("flag") for the script's main loop, and perform the actual processing there.

Don't use potentially blocking commands (like wait_ms, inet.send, inet.recv, inet.connect) in your event handlers ! Invoking such commands from within the event handler increases the risk of abnormal termination of the handler as explained above (after 200 ms).

Similar restrictions also apply to calling the script from the display interpreter.
If the time specified above is not sufficient for your event handler, and *staying in the handler for so long is unavoidable*, the maximum time spent in the handler can be prolonged by feeding a

watchdog in the handler... but this could have the side effects already mentioned before (sluggish response to user input, protocol timeouts, etc).

10.1 3.11.1 Low-level system event handlers

To react on, or even *intercept*, low-level system messages, define one or more of the following handlers in your script.

```
func OnKeyDown( int keyCode )     // a 'normal' key has just been
pressed
func OnKeyUp( int keyCode )       // a 'normal' key has just been
released
func OnEncoderButton(int button) // rotary encoder button pressed
or released
func OnEncoderDelta( int delta ) // rotary encoder position
changed by 'delta' steps
func OnPenDown( int x, int y)     // pen has just been pressed on
touchscreen
func OnPenUp( int x, int y)       // pen has just been released
from touchscreen
func OnPenMove( int x, int y)     // pen coordinate (on
touchscreen) changed, WHILE pen down
func OnGesture( int gestureCode, int gestureSize ) // touchscreen
gesture finished, pen up again
proc OnPageLoaded( int iNewPage, int iOldPage ) // a new display
page was loaded (from FLASH)
```

The above handlers don't need to be registered. They will automatically be called (when implemented in the script), with the function parameters telling the script 'what exactly' has happened (for example, which key has been pressed or released, or the touchscreen coordinate, etc).

For some other kinds of events, arbitrary handler names can be used, for example CAN-Receive- and Timer- events. But even in those cases, we suggest to use consistent names beginning with "On", to tell event handlers from 'normal' functions in the script language.
A few examples:

```
func OnCAN_ID123( tCANmsg ptr pRcvdCANmsg) // CAN-Empfangs-Handler (activated by
can_add_id)
func OnTimer1( tTimer ptr pTimer)          // Timer-Event-Handler (started by
setTimer)
```

The handler may return an integer value of 0 (zero) to let the default message handler process this event as usual; or 1 (one) which means "I have handled this event in my script, and don't want to let the system handle it". This way, the normal keyboard processing can be (almost) completely disabled by returning 1 (1 = "message has been handled").

Note:

If a user-defined event handler doesn't use the 'return <value>' statement, the function returns with an integer value of zero. This means, if an event handler in the script language doesn't return with an explicit value, the event will be handled by the system as usual (which means the event-message will not be suppressed).

A few simple examples for event handlers can be found in the 'EventTest' application.

10.2 3.11.2 Mid-level event handlers (events from visible controls on a UPT display page)

If a system message was not intercepted by a low-level event handler (see previous chapter), it may be propagated to the next level of event handling. This is, in most cases, related to the visible control elements (like buttons, menu items, edit fields, etc) on the current display page.

`func` **`OnControlEvent`**`(` int event, int controlID, int param1, int param2 `)  // event from a 'visible control element'`

This handler may be called for a number of different events. The first function argument indicates *which event* was detected :

`event` : can be one of the following symbolic contstants (in fact, integer numbers):

`evClick` : "the control element has been clicked, or the enter key was pressed while it was focused"
`evPenDown` : "the touchpen has just been pressed, and the touchscreen coordinate was in the control's client area"
`evPenMove` : "the touchpen has been moved, while pressed within the control's client area"
`evPenUp` : "the touchpen has just been released, and the touchscreen coordinate was still in the control's client area"
`evKey` : "a key was sent to the control, while it had the input focus"

`controlID` : user-defined integer value (a constant) to identify the control which fired the event.
The author strongly suggests to use user-defined constants for these identifiers.
The control-ID (also as symbolic constant, not as a stupid 'magic decimal number') must be entered in the field labelled
'Control ID' in the UPT programming tool's page definition table / display line properties.
If a control element doesn't have an ID (i.e. the 'Control ID' field is empty), it will not fire a control event.
The control-ID can also be used to access the display element from within the event handler, using a statement like
`display.elem_by_id[controlID].xyz` (xyz=component of the display element).

`param1` : first message parameter. The meaning depends on the event-type :
For **`evPenDown`** and **`evPenUp`**, param1 is the touchscreen 'X' coordinate, param2 the 'Y' coordinate,
both converted into client coordinates (x=0, y=0 is the control's upper left corner).
For **`evKey`**, param1 contains the keyboard code of the key sent to the control.
`param2` : second message parameter. The meaning depends on the event-type (eg. 'Y').

An example using 'OnControlEvent' can be found in the 'EventTest' application.

11 3.11.3 Timer Events

As already mentioned in the chapter about 'Other functions and commands', the command **setTimer** starts a periodic timer in the script language.
The optional (third) parameter in the argument list is the name of a **timer event handler**, for example:

```
      var
        tTimer timer1;  // Declaration of a timer instance as a global variable
of type 'tTimer'
      endvar;
         ...
      setTimer( timer1, 100, addr(OnTimer1) ); // Start 'timer1'
         // with an interval of 100 milliseconds, to call 'OnTimer1'
periodically
         ...
```

The (function-) name of the timer event handler can be selected freely. We recommend using a descriptive name with the prefix 'On' (as in all event handlers), to tell handlers from ordinary functions. Of course if a script contains multiple timer event handlers, their names must be unique.

In the script language, timers will fire events *periodically* as long as they are not stopped. Whenever a timer's interval expires, the timer's "expired" flag (a component of struct tTimer) is set in the timer variable. If (as in the example shown above) the name of a timer-event-handler has been speficied, that handler will be called shortly after the **'expired'** flag has been set.
The timer variable will be passed to the timer event handler as a pointer (address), so the event handler can easily access it (to start, or modify its own timer).
For that purpose, the timer event handler must be properly defined (as a function which expects the address of a tTimer object):

```
      func OnTimer1( tTimer ptr pMyTimer )  // periodically called Timer Event
Handler
         local tCANmsg msg;  // declaration of a CAN-Message as a local variable
         msg.id := 0x334;    // set the CAN message ID (and, optionally, the bus
number in the MSBits)
         msg.len:= 8;        // set the CAN data length code (max. 8 bytes = 2
doublewords)
         msg.dw[0] := 0x11223344; // set the first four bytes in the CAN message
data as a 'doubleword' (32 bits)
         msg.dw[1] := 0x55667788; // set the last four bytes in the CAN message
data as a 'doubleword'
         can_transmit( msg ); // send the CAN bus message
         return TRUE; // TRUE = 'the timer event has been processed' (FALSE
would not fire more events)
      endfunc; // end OnTimer1
```

If a timer event handler returns 'TRUE' (1), the 'expired'-flag will be cleared by the system, and the timer event handler will be called ***again*** (when the next interval has expired).

If a timer event handler returns 'FALSE' (0), the 'expired'-flag will NOT be cleared, and the event handler won't be called again (i.e. "single shot" operation).

More examples for timer events (in the script) can be found in chapter 4.

12 3.11.4 CAN Receive Handlers

In addition to the polling-method explained in the chapter about CAN functions (i.e. cyclically calling can_receive from the main loop), *event handler* can be implemented in the script which are automatically called on the reception of certain CAN messages (precisely, handlers called on reception of certain CAN message IDs).
This way, the response time can be significantly reduced (in comparison with the polling method).

The following example shows a simple CAN message handler, and the code to *register* that handler by calling can_add_id(<CAN-ID>, <name of the handler>).

```
      func CAN_Handler_ID123( tCANmsg ptr pRcvdCANmsg )
        // CAN-Receive-Handler for certain 'important' CAN message identifiers.
        // Interrupts the normal script processing, and must to the caller
a.s.a.p. !
        select( pRcvdCANmsg.id )  // Take a look at the CAN message
identifier...
          case 0x123:  // message ID 0x123 (hex)
          case 0x124:  // message ID 0x124 (hex)
          case 0x125:  // message ID 0x125 (hex)
              return TRUE; // 'handled' here; do NOT place this message in the
script's CAN-RX-Fifo
        endselect;
        return FALSE; // did NOT handle this message here; let the system place
it in the script's CAN-RX-Fifo
      endfunc; // end CAN_Handler_ID123
```

A CAN receive handler will be called shortly after the reception of a matching CAN message, which ***interrupts*** the normal script processing (and other functions of the programmable terminal). For technical reasons, this interruption may only require a few dozen milliseconds - see details in the yellow info box in the chapter about event handling in the script language.
While the handler is being executed, the device cannot perform other tasks !
If the event handler (function) doesn't 'voluntarily' return to the caller within 500 milliseconds, its execution will be suspended to keep the device operational. The system (firmware) will assume a return value of zero (0), which means the received CAN message will be passed to the system's default-handler for CAN reception.

Meaning of any CAN receive handler's return value :

- **FALSE** (0) : The handler did *not* process this message.
  The system's default handler (implemented in the firmware) will copy the message into the CAN-receive-buffer,
  from where it can be drained by periodically calling can_receive in the script's main loop.

- **TRUE** (1) : The handler has processed the CAN message.
  The received message shall *not* be placed in the CAN-receive-buffer mentioned above.

A *complete* example with a CAN-receive-handler can be found in the application ScriptTest3.cvt .

13 3.11.5 Advanced message handling functions

At the time of this writing (2011-10-04), the following functions were **not implemented yet**... but planned:

- message.register(<message_id>, <flags>) : registers a certain message to be processed (handled) by the script .
- message.deregister(<message_id>) : de-registers (un-registers) a certain message, i.e. informs the system that the script doesn't want to handle this type of message anymore.
- message.peek(out tMessage msg) : Checks if one of the registered messages is waiting in the message queue. If it is, the message is removed from the queue (and copied into 'msg'), and the function returns 1 (one, TRUE). Otherwise (if there is no message waiting in the queue), the function returns immediately with result 0 (zero, FALSE).
- message.get(out tMessage msg, int iTimeout_ms) : Checks if one of the registered messages is waiting in the message queue. If it is, the message is removed from the queue (and copied into 'msg'), and the function returns 1 (one, TRUE). Otherwise (if there is no message waiting in the queue), the function waits for the arrival of the next message. If, during the specified timeout value (in milliseconds) no message arrives, the function returns with result 0 (zero, FALSE). Besides the 'blocking' (waiting) behaviour, there is no difference between message.peek and message.get !
- message.post(<receiver>, <message_id>, <param1>, <param2>, <param3>) : Places a message in the message queue for the specified 'Receiver', which may be something like a window (future plan !). Note that, unlike message.send, message.post returns immediately --- before the receiver has actually processed the message !
- message.send(<receiver>, <message_id>, <param1>, <param2>, <param3>) : Sends a message to the specified 'Receiver' (future plan) or (with iReceiver=0) to the system's default message handler. Note that, unlike message.post, message.send does not return until the message has actually been processed (in other words, it "blocks" the caller). This is not possible with all message types, especially not with those messages which can only be handled in different tasks, or even interrupt service handlers !

The tMessage structure will be specified here in a future version of this document. Most likely, it will be similar to (but not compatible with) Borland's TMessage type .

The message IDs (not to be confused with "CAN" message IDs !) are defined as constants in the script language. Their prefix depends on the message class. For example, messages beginning with **`wm`**... have a similar purpose like 'windows messages' (even if there is no 'Windows' under the hood of the programmable displays). Some of these messages can be used for interaction between the script program, and the graphic user interface. Most notably:

- wmTouchPenDown, wmTouchPenMove, wmTouchPenUp, wmTouchDblClick": low-level touchscreen messages .
- wmEnterDblClick : double-click with the 'Enter' button, whatever the enter button is (it may be a real key, or the rotary encoder button) .

- wmEncoderBtnDown, wmEncoderBtnUp, wmEncoderDblClick, wmEncoderMovedDelta : low-level rotary button events .

Note: Depending on the 'flags' parameter, specified in the message.register command, you can completely 'intercept' certain message types by the script, so they will not be handled by the system anymore.

< To Be Completed ... >

14 3.12 Keyword List

< Incomplete ! Not all of the following keywords are really implemented yet ! Most likely, functions without a hyperlink have not been implemented in the compiler and/or in the runtime library yet >

Even though the script compiler is not case-sensitive, 'basic' keywords which always exist in the language are written in UPPER CASE in the table below, while non-standard keywords (which only may exist in a few, but not all devices) are written in lower case. Crossed out (~~xyz~~) means 'the keyword may exist in future versions, but was not implemented at the time of this writing'.

See also: Quick Reference, Operators .

Keyword	Argument list or syntax	Return value	Remarks
~~ABS~~	(number)	numeric	returns the absolute (non-negative) value. ABS(-1.23) is 1.23 .
~~ACOS~~	(number)	float	arc cosine. The result (angle) is in radians.
AND	A AND B	integer	boolean AND operator, same as the "C"-compatible operator && . The result is 1 (one, TRUE) if **both** operands are non-zero; otherwise zero (FALSE).
~~ASIN~~	(number)	float	arc sine. The result (angle) is in radians.
addr	(variable)	pointer	Returns the *address* of the specified variable.
append	(dest,source[,index])	-	Appends a string (source) to another string or binary block (dest). Result in 'dest'.
atoi	(string)	integer	"ascii to integer". Converts a decimal string into an integer number
BIT_AND	A BIT_AND B	integer	Performs a bit-wise AND operator. Same as "&" in the "C" language. For example, 5 (101 binary) BIT_AND 3 (011 binary) gives 1 (001 binary).
BIT_OR	A BIT_OR B	integer	Performs a bit-wise OR combination of two operands. Same as "|" in the "C" language. For example, 5 (101 binary) BIT_OR 3 (011 binary) gives 7 (111 binary).

BIT_NOT	(unary operator)	integer	Performs a bit-wise NOT operation on the operand on the right side of this *unary* operator. Also known as 'complement'. Example: BIT_NOT 0xFFFF0000 gives 0x0000FFFF as result. For compatibility with the "C" language, the BIT_NOT operator is the same as the tilde (~) . The bitwise NOT operator is often used to invert bitmasks, as in this example
BytesToFloat	(exp, m2, m1, m0)	float	Combines four bytes into a IEEE 754 32-bit single-precision floating point number. The first 8-bit argument (exp1) contains the sign and most significant 7 bits of the exponent. The last 8-bit argument (mantissa_0) contains the least significant bits of the mantissa.
BinaryToFloat	(32-bit 'DWORD')	float	Almost like 'BytesToFloat', but the four bytes are passed as a single 32-bit 'binary' in *little endian byte order* aka 'Intel format'.
BytesToDouble	(exp1,exp0,m5..m0)	float	Combines eight bytes into a IEEE 754 64-bit double-precision floating point number. The first 8-bit argument (exp1) contains the sign and most significant 7 bits of the exponent. The last 8-bit argument (mantissa_0) contains the least significant bits of the mantissa.
can_receive		integer	Tries to read the next received CAN message from a FIFO. When successful, the message is copied into can_rx_msg, and the result is 1 (one) . Otherwise (empty FIFO), can_rx_msg remains unchanged, and the result is 0 (zero) .
CAN. (..)			Prefix (and namespace) for other CAN-related commands and functions. Not for devices with CANopen.
case	integer CONSTANT		part of a select .. case .. else .. endselect block .
chr	(integer code)		Converts an integer character code (0..255, usually from the "DOS" character set) into a single-character string .
cls			clears the text-mode screen (here: a buffer for multi-line text displays).
~~cos~~			
cop. (..)			Prefix (and namespace) for CANopen-related commands and functions. Only for evices with

			CANopen.
~~DEC~~			(reserved for an optimized 'decrement' operator)
display.xyz			Commands and functions controlling the programmable display
dtInteger, etc			Data type codes, used in combination with the typeof() operator.
endif			end of an **if .. then .. [elif ...] else .. endif** construct
endwhile			end of a **while** - loop
endselect			end of a select .. case construct
endproc			marks the end of a user-defined *procedure*
endfunc			marks the end of a user-defined *function*
elif			part of an **if .. then .. elif .. [elif ..] else .. endif** construct ("else if", eliminates the need for additional *endif*s).
else			part of an **if .. then .. else .. endif** construct, or a select .. case .. **else** .. endselect block (depends on the context it is used in)
EXOR	A EXOR B (binary operator)	integer	This operator performs a bitwise EXCLUSIVE-OR combination of the two operands. Often used to toggle (invert) one or more bits in a bitmask. For example, 5 (0101 binary) EXOR 3 (0011 binary) gives 6 (0110 binary). Notes: • The "^" operator is NOT used as the EXOR operator in the script language ! (A ^ B is reserved for "A power B" in future versions of this language) • There is no BOOLEAN EXOR, because that would be the same as the 'not equal' operator . • The bitwise EXOR operator is typically used to toggle bits, as in this example .
file.			prefix for all file I/O functions
float			data type for floating point values
for			begins a for .. to .. step .. next loop

ftoa	(value, nDigitsBeforeDot, nDigitsAfterDot)		'floating point to ASCII'
~~GOTO~~			stoneage jump instruction, try to avoid whereever possible
~~GOSUB~~			old subroutine calls, try to avoid ...
gotoxy	(x,y)		sets the text output cursor in the specified column (x, 0..79) and row (y, 0..24) .
hex	(value, nDigits)		Converts an integer value into a fixed-length *hex* (hexadecimal) string, with the specified number of digits.
if	(condition)		statement begins an IF .. THEN .. ELSE .. ENDIF construct
in			defines the following argument (in a formal argument list) as "input" for a procedure
~~INC~~			(reserved for an optimized 'increment' operator)
inet.			prefix for socket-based internet functions
int			integer (data type; 32 bit signed integer)
isin	(argument:0...1023)		Fast integer sine function. Input range 0 (~0°) to 1023 (~360°). Output range -32767 to +32767.
itoa	(value, nDigits)		'integer to ASCII'. Converts an integer value into a fixed-length *decimal* string, with the specified number of digits.
~~LEFT~~			
~~LEN~~			
local			defines a few local variables (which exist 'on the stack' until the end of a user-defined function)
~~LOG~~			
~~LN~~			
~~MID~~			
~~MODE~~			
MOD			
next			ends any **for .. to .. step .. next** loop

NOT			boolean 'NOT' operator (negation). Same as the "C"-compatible "!" operator ("unary not"). The result is 1 (one, TRUE) if the input is 0 (zero, FALSE); otherwise (if the input is non-zero), the result is zero.
OR			boolean OR operator, same as the "C"-compatible operator || . The result is 1 (one, TRUE) if **any** of the operands is non-zero; otherwise zero (FALSE).
out			defines the following argument (in a formal argument list) as "output" (or input/output) for a procedure or function .
print			Prints values into a multi-line "text panel" on the current display page .
cPI			constant value "PI" (3.14159....) .
proc			Marks the begin of a user-defined *procedure*
~~POS~~			
ptr			Keyword for a typeless or fixed-type pointer
ramdom	(N)		generates a pseudo-random number between zero and N minus one .
return			Returns from a user-defined function (with a "value"). In older scripts, returns from a 'subroutine' invoked with 'gosub' (deprecated, without a value).
repeat			begins a REPEAT..UNTIL loop. This kind of loop is executed *at least once* .
REM			begins a remark in BASIC. Better use the double slash // - same effect
RGB	(red, green, blue)		Composes a colour from red, green, and blue components
~~RIGHT~~			
select	(integer expression)		begins a select .. case .. else .. endselect block .
setcolor			Sets the foreground- and background colour for output on the text screen .
~~SIN~~			
SHL	(binary operator)		Bitwise shift left. Example N := N SHL 8; //

			multiplies N by 256

SHR	(binary operator)		Bitwise shift right. Example N := N SHR 2; // divides N by four .
			Note: SHR is considered an 'arithmetic' shift right. Negative numbers remain negative, and positive numbers remain positive. Thus, SHR expands the sign from bit 31, and 0x80000000 SHR 31 gives the result 0xFFFFFFFF (not 0x00000001) .

~~SQRT~~			
~~STR~~			
step			defines the counter's stepwidth in a **for .. to .. step .. next** loop
stop			Stops execution of the script. Useful for debugging.
string			data type for a 'string' of characters
system	.component-name		access a few 'system' variables (current timestamp, etc), or invokes functions like system.beep
~~TAN~~			
tCANmsg			data type name for a 'CAN message'. This type is also used for global variables like 'can_rx_msg' .
time			Date and Time conversions
tMessage			data type name for a system message (can be used for event / message handling in the script program).
to			defines the counter's end value in a **for .. to .. step .. next** loop
trace	.print, .enable, ..		Trace History control
tscreen	.component-name		text-screen buffer object
tScreenCell			data type name for a 'text screen cell'
typedef			defines a new data type (usually a structure composed of basic data types)
typeof			Retrieves the *momentary* data type of the specified variable (declared as 'anytype'). The result is usually a data type constant, for example dtInteger.

until	(end criterion)		ends a REPEAT..UNTIL loop
wait_ms	(milliseconds)		waits for the specified number of milliseconds, before continuing with the next script instruction
while			

See also:

- list of built-in constants
- list of built-in data type names (which are reserved keywords, too)
- list of built-in operators (some of them are also reserved names / keywords, no 'special characters' !)

3.13 Error messages

The following list is very incomplete (due to a lack of time..) ! Most error messages should speak for themselves.

- syntax error
  An error has occured, usually during compilation, which cannot be further diagnosed by the compiler.
- missing argument
  The argument list of a function or procedure call contains less arguments than expected.
- missing left parenthesis
- missing right parenthesis
- missing LEFT square bracket ([ )
- missing RIGHT square bracket ( ])
- missing operand
- type conflict
- division by zero
- illegal value
- function unknown
- function permanently unavailable
- function temporarily unavailable
- illegal array index or similar
- missing component
- unknown component
- function failed
- bad array subscript
- illegal pointer or reference
- comma or closing parenthesis expected
- expecting a semicolon
- expecting a comma
- name expected
- var-name expected
- expecting an assignment
- expecting a data type

- expecting an integer value
- undefined variable
- out of memory
- structure or block too large
- illegal channel number
- label not found
- return without gosub
- call stack overflow
- call stack underflow
- call stack corrupted
  Occurrs, for example, when a user-defined function tries to return to the caller but finds no valid return address on the stack (value exceeds program memory size, or negative address). In fact, the "call stack" is the same stack used for RPN calculations, so if a calculation illegally overwrites a return address, such errors may occur.
- name or symbol too long
  Names of variables, functions, data types, constants, etc are all limited to 20 characters.
- subscript or indirection too long
  Applies to arrays and/or nested structures.
  If, for example, A is a one-dimensional array, A[1][2] is illegal ("too many array indices" in this case).
- bad input
  An input-function, or string parser, couldn't handle the input (for example, could not convert the input characters into a number).
- 'for' without 'next'
- 'next' without 'for'
  Often occurs as a subsequent error when there was a problem in the matching 'for' (error disappears after fixing the problem in the 'for' statement).
- 'else' without 'if'
- 'endif' without 'if'
- 'case' without 'select'
- 'endselect' without 'select'
- 'endwhile' without 'while'
- 'until' without 'repeat'
- no loop to exit from
- only callable from SCRIPT
- simple variables only
- variable or element is READ-ONLY
- unknown script command
- function not implemented yet
- RPN eval stack overflow
- RPN eval stack underflow
- illegal code pointer
- illegal sub-token after opcode
- cannot use as LVALUE (in assignment)
- not an allowed ARRAY type
  The element left of an array subscript cannot be accessed like an array.
- ARRAY type mismatch (dimensions, etc)

- name already defined
  The name you tried to use in a variable declaration, type definition, or similar is already in use.
- missing 'struct' or 'endstruct'
- internal error - SORRY !
  If you ever encounter this error, please report this error to the developer (Wolfgang Büscher), along with the sourcecode of the script which was causing it.
- unknown error code (< number >)
  An error code has occurred for which there is no entry in the error message table yet. Please report this error to the developer, along with the error number.

4. Examples

The programming tool's installer contains a few 'tests' (used during development) and 'examples' (planned).
The following examples are part of programming tool's installer:

script_demos\ScriptTest1.cvt
A very basic example, used to test program flow controls (especially select..case), operators, built-in keywords, etc.

script_demos\ScriptTest2.cvt
A speed test for the script's runtime function. Calculates the value of PI (3.14159) using an iteration loop.

script_demos\ScriptTest3.cvt
A slightly more advanced test application. Used during development to test arrays, type definitions, scrollable text, and CAN-bus functions.

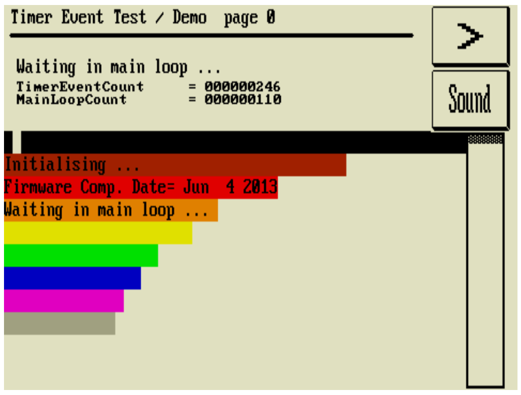
script_demos\DisplayTest.cvt
Test application to control some display elements via script, like display.menu_mode, display.menu_index. Also calls a very simplistic user-defined procedure ("GoToNextField") from a graphic button:

```
//-------------------------------------------------------------
proc GoToNextField // Called from the display (on button)
  display.menu_mode  := mmNavigate; // switch to "select"
(navigate) mode
  display.menu_index := (display.menu_index+1) % 8; // switch
to next field
endproc; // end GoToNextField
```

script_demos\TimerEvents.cvt
Test and demo for Timer-Events. This script uses an array of timers, programmed with different cycle times:

```
const     // define a few constants...
   C_NUM_TIMERS = 10;         // number of simultaneously running timers
   int ResistorColours[10] =  // ten 'colour codes' [0..9]:
    { clBlack, clBrown, clRed,     clOrange, clYellow, // [0..4]
      clGreen, clBlue,  clMagenta, clDkGray, clLtGray  // [5..9]
    };
endconst;
   ...
typedef
   tMyTimerControl = struct
      int   iEventCount;
      float fltMeasuredFrequency;
   endstruct;
endtypedef;
   ...
   var // declare a few GLOBAL variables...
     tTimer MyTimer[C_NUM_TIMERS]; // an ARRAY of timers for the stress-
test
     tMyTimerControl MyTimerCtrl[C_NUM_TIMERS]; // an array of self-defined
'timer controls'
   endvar;
      ...
   // Start the timers for the timer 'stress test'
   for i:=0 to #(C_NUM_TIMERS-1)
     MyTimer[i].user := i;  // use the index as 'user defined ID' for this
timer
     setTimer( addr(MyTimer[i]), 37+20*i/*ms*/, addr(OnMyTimer) );
   next;
      ...
   //------------------------------------------------------------------
   func OnMyTimer( tTimer ptr pMyTimer )  // another TIMER EVENT HANDLER...
     // Shared by  timers which run at different intervals.
     // The activity of each of these timers is visualized
     //     as a horizontal coloured bar on a text panel .
     local int i,x,y;
     local tMyTimerControl ptr pCtrl;
     debugTimer := pMyTimer[0]; // copy the argument into a global variable
(for debugging/"Watch")
     TimerEventCount := TimerEventCount + 1; // global counter for ALL
timer events
     i := pMyTimer.user;        // user defined index of this timer, here:
i = 0..9
     pCtrl := addr(MyTimerCtrl[i]); // address of a user-defined 'timer
control' struct
     x := pCtrl.iEventCount % tscreen.vis_width;
     y := i;
     tscreen.cell[y][x].bg_color := ResistorColours[i];
     tscreen.cell[y][x+1].bg_color := clWhite;
     tscreen.modified := TRUE;
     pCtrl.iEventCount := pCtrl.iEventCount + 1; // count the events fired
by THIS timer
      // ...
     return TRUE; // TRUE = 'the event has been handled here' (FALSE would
not fire more timer events)
   endfunc; // OnMyTimer()
```

On each timer event, a counter for that timer is incremented, and the counter value is displayed as a horizontal colour bar on a Text-Panel :



```
Timer Event Test / Demo  page 0

Waiting in main loop ...
TimerEventCount      = 000000246
MainLoopCount        = 000000110

Initialising ...
Firmware Comp. Date= Jun  4 2013
Waiting in main loop ...
```

(Screenshot from the 'Timer Event' test application)

The upper bar shows the 'fastest running' timer (index 0, colour code black), the lower bar shows the slowest timer (here: index 8, colour code gray).

To measure the timing jitter, an additional timer event is used, which cyclically transmits a CAN message:

```
  //--------------------------------------------------------------------
  func OnCANtxTimer( tTimer ptr pMyTimer )  // a TIMER EVENT HANDLER...
    local tCANmsg msg;  // use LOCAL variables (not globals) in event
handlers !
    msg.id := 0x335;    // set CAN message identifier for transmission
    msg.len:= 8;        // set CAN data length code (max. 8 bytes = 2
doublewords)
    msg.dw[0] := 0x11223344; // set four bytes in a single doubleword-move
(faster than 4 bytes)
    msg.dw[1] := 0x55667788; // set the last four bytes in the 8-byte CAN
data field
    can_transmit( msg ); // send the CAN message to the bus
    return TRUE; // TRUE = 'the event has been handled here' (FALSE would
not fire more timer events)
  endfunc;
```

The timer for this event handler is started in the initialisation part of the script as follows:

```
  // Start another timer for periodic CAN transmission .
  // Jitter can be checked with a CAN bus analyser.
  setTimer( CANtxTimer, 100/*ms*/, addr(OnCANtxTimer) ); // OnCANtxTimer
called every 100 ms
```

When tested on an MKT-View III, a jitter of approximately +/- 5 milliseconds could be observed with a CAN bus tester. Most of this jitter is caused by the *cooperative*, not *preemptive*, multitasking within the script language. The jitter *may* be reduced in future versions of the device firmware (2013-06-05).

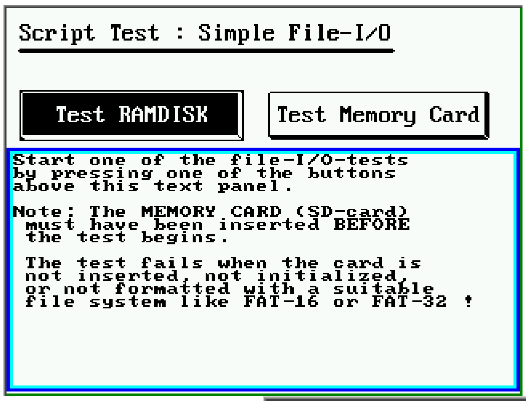
script_demos\FileTest.cvt

Test- and demo application for the file I/O functions. Different tests can be started by the graphic buttons on the first display page:
'Test RAMDISK' writes and reads a file on the device's RAMDISK,
'Test Memory Card' uses the SD memory card for the same test.
The function 'Test File Access' (written in the script language) is used for both storage media. Here is a *shortened* version of it:

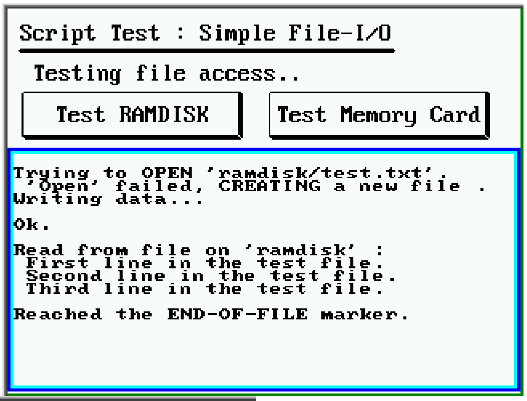
```
//-----------------------------------------------------------------
func TestFileAccess( string pfs_path )
  // Part of the test program for file I/O functions . Taken from
'FileTest.cvt' .
  // [in]  pfs_path : path for the pseudo-file-system like "ramdisk" or
"memory_card"
  local int fh;        // file handle
  local int i;
  local string fname;
  local string temp;

  // Build a complete filename, with a path:
  fname := pfs_path+"/test.txt";

  // First try to OPEN the file (but don't try to CREATE, i.e. OVERWRITE
it):
  fh := file.open(fname,O_RDWR); // try to open existing file, read- AND
write access
  if( fh>0 ) then
    file.seek(fh, 0, SEEK_END ); // Set file pointer to the END of the file
    // write a separator between the 'old' and the 'new' part of the file:
    file.write(fh,"\r\n---- data appended to file ----\r\n");
  else  // file.open failed, so try to CREATE a 'new' file:
    fh := file.create(pfs_path+"/test.txt",4096);  // create a file, with
pre-allocation
  endif;
  if( fh>0 ) then    // successfully opened or created the file ?
    file.write(fh,"First line in the test file.\r\n");
    file.write(fh,"Second line in the test file.\r\n");
    file.write(fh,"Third line in the test file.\r\n");
    file.close(fh);
  else               // neither file.open nor file.create were successful:
    print( "\r\nCould not open or create a file !" );
    return FALSE;
  endif;

  // After writing and closing the file (above), open it again, and READ
the contents:
  fh := file.open(pfs_path+"/test.txt", O_RDONLY);  // try to open the file
for READING
  if( fh>0 ) then    // successfully created the file ?
    print( "\r\nRead from file on '",pfs_path,"' :" );
    while( ! file.eof(fh) ) // repeat until the end of the file.......
      temp := file.read_line(fh); // read one line of text from the file
      print( "\r\n ", temp );      // dump that line to the text panel
      Progress := Progress+1;
    endwhile;
```

```
      print( "\r\nReached the END-OF-FILE marker." );
      file.close(fh);
    else
      print( "\r\nCould not open file for reading !" );
      return FALSE;
    endif;
    return TRUE;
endfunc; // TestFileAccess
```

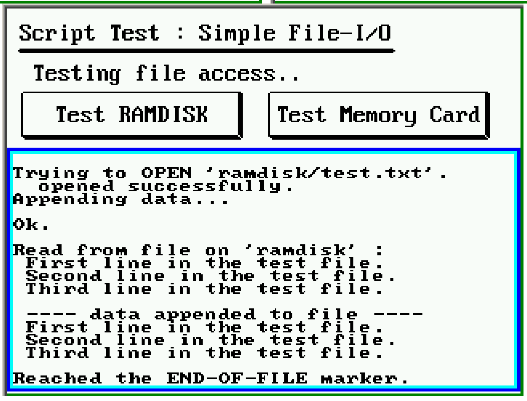
With some additional output from the 'print' command, and pressing the 'Test RAMDISK' button, the program produced the following output on a text panel:



```
Start one of the file-I/O-tests
by pressing one of the buttons
above this text panel.

Note: The MEMORY CARD (SD-card)
 must have been inserted BEFORE
 the test begins.

 The test fails when the card is
 not inserted, not initialized,
 or not formatted with a suitable
 file system like FAT-16 or FAT-32 !
```



```
Trying to OPEN 'ramdisk/test.txt'.
 'Open' failed, CREATING a new file .
Writing data...
Ok.
Read from file on 'ramdisk' :
 First line in the test file.
 Second line in the test file.
 Third line in the test file.
Reached the END-OF-FILE marker.
```



```
Trying to OPEN 'ramdisk/test.txt'.
  opened successfully.
Appending data...
Ok.
Read from file on 'ramdisk' :
 First line in the test file.
 Second line in the test file.
 Third line in the test file.
 ---- data appended to file ----
 First line in the test file.
 Second line in the test file.
 Third line in the test file.
Reached the END-OF-FILE marker.
```

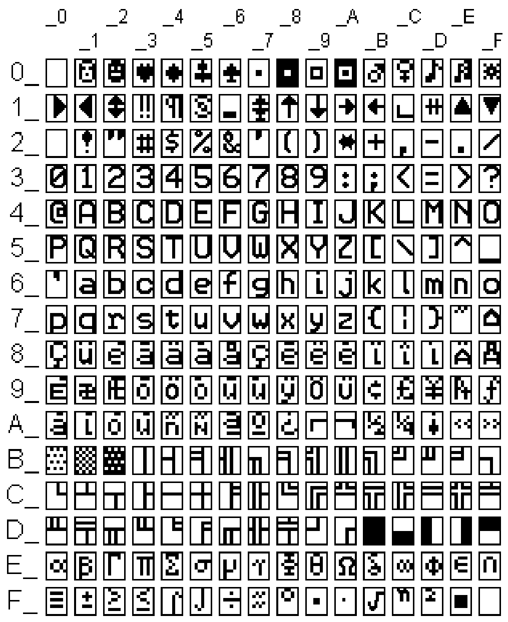
script_demos\TScreenTest.cvt

Test application for the text-screen buffer, with procedures to draw lines and frames in the text-mode screen buffer ("tscreen") .
This file was also used as a first test for local variables, parameter passing in procedures, and recursive calls during development .

```
//--------------------------------------------------------------
proc FillScreen1(string sFillChar)
  // PROCEDURE to fill the screen with a colour test pattern .
  LOCAL int X,Y,old_pause_flag;
  old_pause_flag := display.pause;
```

```
 display.pause  := TRUE; // disable normal screen output
 for Y:=0 to tscreen.ymax
  for X:=0 to tscreen.xmax
   gotoxy(X,Y);
   setcolor( clWhite,
     RGB((11*(X+Y))&255, (9*(Y-X))&255, (3*X)&255 ) );
   print( sFillChar ); // print a single character
  next;
 next;
 display.pause:=old_pause_flag; // resume display output ?
endproc; // end FillScreen1()
```

Furthermore, TScreenTest .cvt demonstrates how special DOS characters (from "codepage 437" or "codepage 850") can be used to draw lines, boxes, and grids on a text screen. This demo also contains a very simple 'video game' which polls the keyboard to steer a worm (or snake) through a maze. A similar principle can be used in your application to realize advanced animated graphics, using the special "graphic" characters from a DOS compatible font like the one shown below:


_0 _1 _2 _3 _4 _5 _6 _7 _8 _9 _A _B _C _D _E _F
0_ 1_ 2_ 3_ 4_ 5_ 6_ 7_ 8_ 9_ A_ B_ C_ D_ E_ F_
DOS characters, codepage 437

script_demos\LoopTest.cvt

Demonstrates various loop commands, like for-to-next . Contains a simple 'animated' colour text demo using loops, gotoxy, color, RGB, the print command, and the display.pause flag to prevent the display from being updated at the 'wrong' time (here to avoid flicker while filling the text-screen buffer with new characters) .

script_demos\TimeTest.cvt

Test application for the time-conversion functions like time.date_to_mjd and time.mjd_to_date .
Also shows how to split a 'Unix Time' into years, months, days, hours, minutes, and seconds.

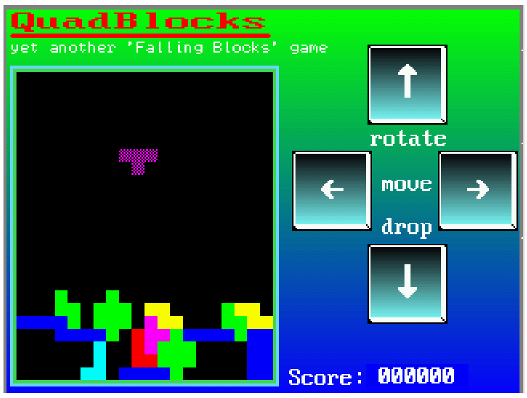
script_demos\StringTest.cvt
Test application for string functions like strlen(), strpos(), substr(), etc.

script_demos\StructArrayTest.cvt
Test application for an **array** of user-defined **structures**, with each struct containing **integers, floating point values, and strings** .
The array-filling loop also served as a simple benchmark during development in October 2010 .

script_demos\PageMenu.cvt
This demo application builds a menu showing all the existing display pages (in the application) in a menu, and allows jumping to the selected page (in the menu).

script_demos\EventTest.cvt
Test/demo to handle low-level events ("system events") in the script language.

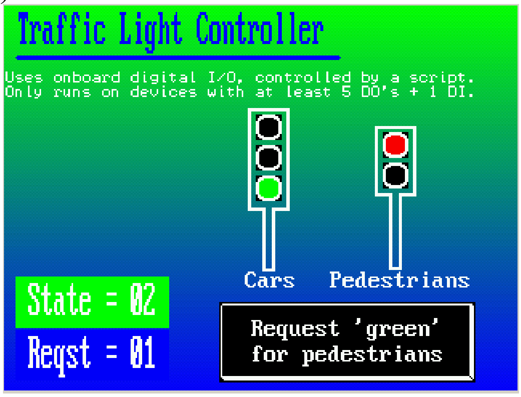
script_demos\quadblox.cvt
Test application for **two-dimensional array variables** and **two-dimensional array constants** . Also shows how to poll the keyboard, or events fired by programmable buttons on a certain display page. This demo is actually a simplified implementation of a once-famous "puzzle game with falling blocks", which we don't call by its original name to avoid copyright hassle. The cursor keys may be emulated with the graphic buttons (for devices with touchscreen) on the right side of the screen shown below.



The "QuadBlocks"-demo was designed for a 320*240 pixel screen, but it can automatically resize itself for displays with 480*272 pixels using the functions tscreen.vis_width and tscreen.vis_height . For devices with 240*320 pixels (aka "portrait mode" screen), the script switches to a different display page than the for "landscape" mode, by checking the horizontal screen resolution (display.pixels_x, which may be 128, 240, 320, or 480 pixels, depending on the target hardware).

script_demos\TrafficLight.cvt

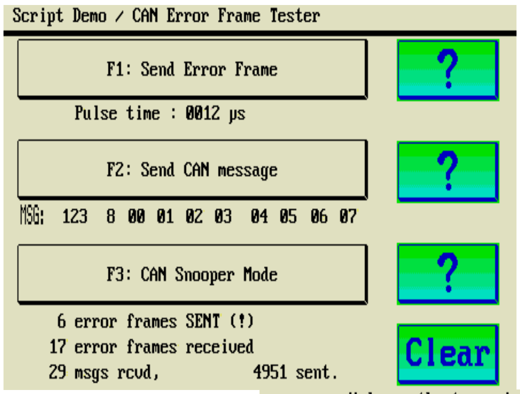
Simple 'traffic light controller'. Demonstates the use of the onboard digital I/O lines (which only exist in a few devices).


Traffic Light Controller
Uses onboard digital I/O, controlled by a script.
Only runs on devices with at least 5 DO's + 1 DI.
Cars
Pedestrians
State = 02
Reqst = 01
Request 'green' for pedestrians

Pedestrians can request 'green' by pushing a button, connected to the first digital input. Alternatively, a graphic button on the touchscreen can be used for this.
Three digital outputs are used for the 'cars' (red, yellow, green), two digital outputs for the pedestrians (red alias "don't walk", green alias "walk").
The traffic light states are also displayed on the screen, using bitmaps which change their colours depending on the current states of the digital outputs.
It also shows how to change the colour of a certain display element (on the current display page) through the script, using display.elem .

script_demos\ErrFrame.cvt

Specialized CAN test, allows to send (!) and receive (count) error frames. On each received CAN error frame, this application produces an acoustic signal - which may be a helpful testing utility. Ideally, error frames do not occurr on a CAN (Controller Area Network) - but in practise, they do happen. Some poorly designed devices used to send a dominant bit sequence (six or more dominant bits) during startup. This utility helps to spot such errors. It can also be used to 'probe' a network: Send an error frame into the CAN, and (if there's "someone else" on the bus), you will receive one error frame in response. Otherwise, you know this CAN-bus is "dead".
The length of the error frame (in microseconds) can be modified in the edit field labelled 'Pulse Time'. The default, 12 microseconds, will only give an error frame if the CAN bus runs at 1000 or 500 kBit/second. To cause an error frame for lower bitrates, increase the number of microseconds.

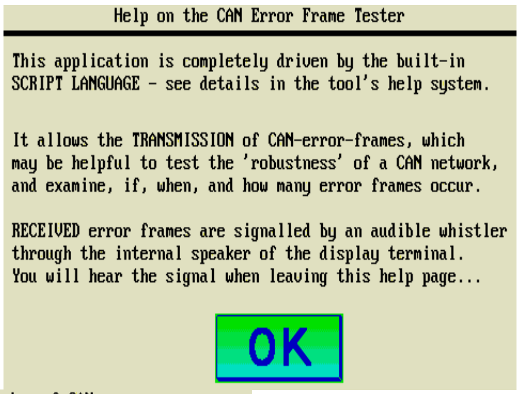


Script Demo / CAN Error Frame Tester

Pulse time : 0012 µs

MSG: 123 8 00 01 02 03 04 05 06 07

6 error frames SENT (!)
17 error frames received
29 msgs rcvd, 4951 sent.

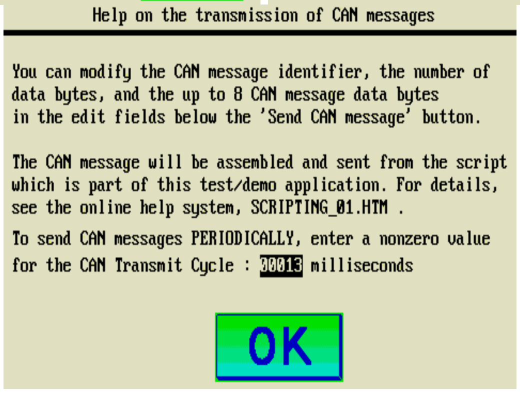


Help on the CAN Error Frame Tester

This application is completely driven by the built-in SCRIPT LANGUAGE - see details in the tool's help system.

It allows the TRANSMISSION of CAN-error-frames, which may be helpful to test the 'robustness' of a CAN network, and examine, if, when, and how many error frames occur.

RECEIVED error frames are signalled by an audible whistler through the internal speaker of the display terminal. You will hear the signal when leaving this help page...



Help on the transmission of CAN messages

You can modify the CAN message identifier, the number of data bytes, and the up to 8 CAN message data bytes in the edit fields below the 'Send CAN message' button.

The CAN message will be assembled and sent from the script which is part of this test/demo application. For details, see the online help system, SCRIPTING_01.HTM .

To send CAN messages PERIODICALLY, enter a nonzero value for the CAN Transmit Cycle : 00013 milliseconds

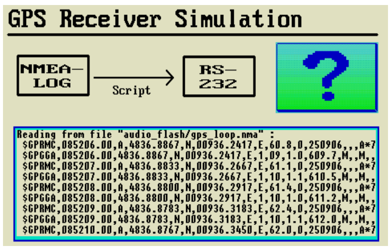
To put the CAN network under 'sufficient stress', this application can also transmit normal CAN messages to the bus.
The format is:

CAN message ID (hex), number of data bytes (0..8), and up to 8 bytes (also hexadecimal).

Transmission can be initiated manually (button 'Send CAN message' in the 1st screenshot above) or periodically, using a timer event with an adjustable interval (see 3rd screenshot above).

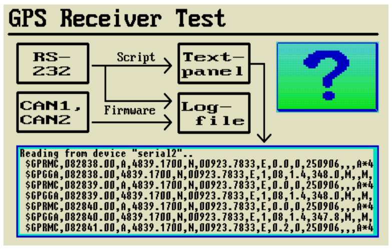
script_demos\SerialPt.cvt ; script_demos\GpsRcv01.cvt ; script_demos\GpsSim01.cvt

Various tests for the serial ports, most of them for the MKT-View II (which, unlike most other devices, has *two* serial ports). The application 'GpsSim01.cvt' was used to simulate a GPS receiver by reading lines from an NMEA log file from a text file (line by line), and sending them through the serial port. The application 'GpsRcv01.cvt' is more or less the counterpart: It was used on a second MKT-View II (both connected through a modified "Null-Modem" cable) to display the received NMEA strings on a text panel.
Note: Because the serial port is accessed through the file I/O functions, these demos only work if the *extended script functions* are unlocked !



```
Reading from file "audio_flash/gps_loop.nma" :
 $GPRMC,085206.00,A,4836.8867,N,00936.2417,E,60.8,0,250906,,,A*7
 $GPGGA,085206.00,4836.8867,N,00936.2417,E,1,09,1.0,609.7,M,,M,,
 $GPRMC,085207.00,A,4836.8833,N,00936.2667,E,61.1,0,250906,,,A*7
 $GPGGA,085207.00,4836.8833,N,00936.2667,E,1,10,1.1,610.5,M,,M,,
 $GPRMC,085208.00,A,4836.8800,N,00936.2917,E,61.4,0,250906,,,A*7
 $GPGGA,085208.00,4836.8800,N,00936.2917,E,1,10,1.0,611.2,M,,M,,
 $GPRMC,085209.00,A,4836.8783,N,00936.3183,E,62.4,0,250906,,,A*7
 $GPGGA,085209.00,4836.8783,N,00936.3183,E,1,10,1.1,612.0,M,,M,,
 $GPRMC,085210.00,A,4836.8767,N,00936.3450,E,62.0,0,250906,,,A*7
```

-------- ("Null-Modem cable") -------->



```
Reading from device "serial2"..
 $GPRMC,082838.00,A,4839.1700,N,00923.7833,E,0.0,0,250906,,,A*4
 $GPGGA,082838.00,4839.1700,N,00923.7833,E,1,08,1.4,348.0,M,,M,
 $GPRMC,082839.00,A,4839.1700,N,00923.7833,E,0.0,0,250906,,,A*4
 $GPGGA,082839.00,4839.1700,N,00923.7833,E,1,08,1.4,348.0,M,,M,
 $GPRMC,082840.00,A,4839.1700,N,00923.7833,E,0.0,0,250906,,,A*4
 $GPGGA,082840.00,4839.1700,N,00923.7833,E,1,08,1.4,347.8,M,,M,
 $GPRMC,082841.00,A,4839.1700,N,00923.7833,E,0.2,0,250906,,,A*4
```

To access the serial port from the script language, use the device names "serial1" (= the first serial port) or "serial2" (= the 2nd serial port, in the MKT-View II this port is decicated for the GPS receiver). Here is a sourcecode snippet from the 'GPS receiver' demo (GpsRcv01.CVT) :

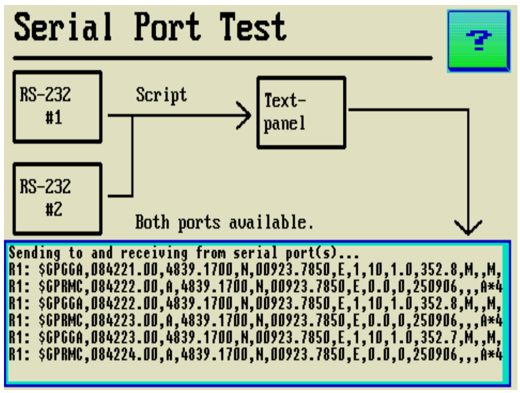
```
// Try to open the second serial port "like a file" .
// In the MKT-VIEW II, device "serial2" is the GPS port.
// Note: Do NOT modify the serial port's baudrate here.
// The system has already set it, according to the
// 'System Setup' / 'GPS Rcv Type' (4k8, 9k6, ...).
hSerial := file.open("serial2");
if( hSerial>0 ) then // successfully opened the GPS port
   display.PortInfo := "Port2";
else // Could not open the 2nd serial port !
     // This happens on a PC (which has no dedicated GPS
port).
     // Try the FIRST serial port instead, with a fixed
baudrate:
   hSerial := file.open("serial1/9600");
   display.PortInfo := "Port1";
endif;
if( hSerial<=0 ) then // could not open the serial port ?
   print( "\r\nCouldn't open the serial port !" );
   stop;
endif;

print( "Reading from
device \"",file.name(hSerial),"\"..\r\n" );
while(1) // endless loop to read and process received data...
     // Read the next bytes from the serial port
     // (not necessarily a complete LINE) :
   temp := file.read_line(hSerial);
   if( temp != "" ) then // something received ?
```

```
        // Dump the received character(s) to the text panel..
        if( tscreen.cy >= tscreen.vis_height ) then
             gotoxy(0,1); // wrap around
        endif;
        print( " ", temp );
        clreol;
        print( "\r\n" );
        clreol; // clear the next line (to show last entry)
    else // nothing received now ..
        wait_ms(50); // .. let the display program work
    endif;
endwhile;
```

The script in the 'Serial Port Test' application (programs\script_demos\SerialPt.CVT) also uses the file-I/O API to open *both* serial ports (which is possible in the MKT-View II), and reads anything received from both ports line-by-line. In this case, the script *tries to* open both serial ports with the same, fixed bitrate:

```
// Try to open the second serial ports "like files" .
// In the MKT-VIEW II, device "serial2" is the GPS port.
// To open the serial port with a fixed baudrate,
// append it after the device name as below .
hSerial1 := file.open("serial1/9600"); // open 1st serial port

hSerial2 := file.open("serial2/9600"); // open 2nd serial port
```

The received strings are dumped to a text panel. The LCD may look like this:



```
Both ports available.
Sending to and receiving from serial port(s)...
R1: $GPGGA,084221.00,4839.1700,N,00923.7850,E,1,10,1.0,352.8,M,,M,
R1: $GPRMC,084222.00,A,4839.1700,N,00923.7850,E,0.0,0,250906,,,A*4
R1: $GPGGA,084222.00,4839.1700,N,00923.7850,E,1,10,1.0,352.8,M,,M,
R1: $GPRMC,084223.00,A,4839.1700,N,00923.7850,E,0.0,0,250906,,,A*4
R1: $GPGGA,084223.00,4839.1700,N,00923.7850,E,1,10,1.0,352.7,M,,M,
R1: $GPRMC,084224.00,A,4839.1700,N,00923.7850,E,0.0,0,250906,,,A*4
```

script_demos\InetDemo.cvt

Test, demo, and example application for the Internet functions in the script language. Implements a simple TCP/IP-based client and server. Also runs in the simulator, integrated in the programming tool, if you allow the program to 'act as a server' in the windows security center / personal firewall.

script_demos\MultiLanguageTest.cvt

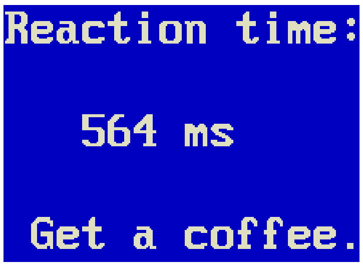
This application started as a test program for various script language extensions. It uses the file I/O functions to read textfiles stored in any of the device's FLASH memory files (using the UPT's pseudo-file system), and a user-defined function ("GetText") which is called from the display in a backslash sequence to retrieve a text (string) in one of many selectable languages.
It also shows how to invoke script procedures from display interpreter commandlines.

script_demos\OperatorTest.cvt

This application contains a test script for some 'advanced' operations. It was used during software development to check various operators (mostly the assignment operator) and other new functions.

script_demos\ReactionTest.cvt

This application implements a simple 'reaction time test' for a human operator. The script first waits for a random time (between 0.5 and 5 seconds), then changes the display colour, and measures the time until the operator hits any key, or hits the touchscreen surface, or turns/rotates the rotary encoder knob.
Depending on your reaction speed, the program suggests how to proceed:



```
Reaction time:

   564 ms

 Get a coffee.
```

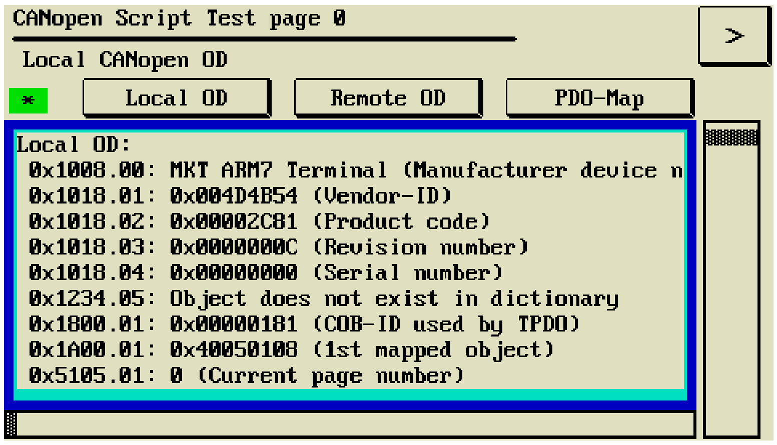
script_demos\TraceTest.cvt

This application contains a short script to test the Trace-Historie, using the Commands to control the Trace History from chapter 3.10 .
This includes ("but it not limited to"):

- Appending own messages via script to the trace history (trace.print)
- Automatically stopping the trace history via script
- Excluding certain CAN messages from the trace history (trace.can_blacklist)
- Showing the contents of the trace history on a text panel (trace.entry[n])
- Clearing the trace history *by the operator* (via graphic button / touchscreen)
- Saving the trace history as a text file on the device's memory card

script_demos\CANopen1.upt

Only for the 'UPT Programming Tool II' and for devices with integrated CANopen protocol stack.



```
Local OD:
 0x1008.00: MKT ARM7 Terminal (Manufacturer device n
 0x1018.01: 0x004D4B54 (Vendor-ID)
 0x1018.02: 0x00002C81 (Product code)
 0x1018.03: 0x0000000C (Revision number)
 0x1018.04: 0x00000000 (Serial number)
 0x1234.05: Object does not exist in dictionary
 0x1800.01: 0x00000181 (COB-ID used by TPDO)
 0x1A00.01: 0x40050108 (1st mapped object)
 0x5105.01: 0 (Current page number)
```

The graphic buttons 'Local OD', 'Remote OD', and 'PDO-Map' above the text panel launch different subroutines in the script. The scrollers on the right side and below the text panel can be operated via touchscreen to scroll the visible part of the larger 'virtual text screen' vertically and horizontally.
Here is a slightly shortened variant the sourcecode of the 'PDO-Mapping Test', which reprograms the mapping of the first transmit-PDO:

```
//-------------------------------------------------------------------
proc TestPdoMapping // demo to 'reprogram' this device's own PDO mapping
  // 'Reprogram' a CANopen slave's PDO mapping table...
  // How to do it CORRECTLY (quoted from CiA 301, V4.2.0, page 142..143):
  // > The following procedure shall be used for re-mapping,
  // > which may take place during the NMT state Pre-operational
  // > and during the NMT state Operational, if supported:
  // >  1. Destroy TPDO by setting bit valid to 1b of sub-index 01h
  // >     of the according TPDO communication parameter.
  // >  2. Disable mapping by setting sub-index 00h to 00h.
  // >  3. Modify mapping by changing the values of the corresponding sub-
indices.
  // >  4. Enable mapping by setting sub-index 00h to the number mapped
objects.
  // >  5. Create TPDO by setting bit valid to 0b of sub-index 01h
  // >     of the according TPDO communication parameter.
  local dword dwCommParValue; // a 32-bit unsigned integer variable, aka
dword

  cop.error_code := 0; // Clear old 'first' error code (aka SDO abort code)
.
   // If the following CANopen commands work as planned, cop.error_code
remains zero.
   // If something goes wrong, cop.error_code could tell us what, and why
it went wrong.
   // In the original script (in script_demos/CANopen1.upt), it is checked
after each obd-access.
```

```
  dwCommParValue := cop.obd(0x1800,0x01); // save original value of PDO
communication parameter
  cop.obd(0x1800,0x01) := dwCommParValue | 0x80000000;  // make 1st
transmit PDO invalid (CiA: "Destroy TPDO"); set bit 31
  cop.obd(0x1A00,0x00) := 0x00;        // clear TPDO1 mapping table (CiA:
"Disable mapping"..)
  cop.obd(0x1A00,0x01) := 0x40050108;  // 1st mapping entry: map object
0x4005, subindex 1, 8 bits
  cop.obd(0x1A00,0x02) := 0x50010108;  // 2nd mapping entry: map object
0x5001, subindex 1, 8 bits
  cop.obd(0x1A00,0x03) := 0x51050108;  // 3rd mapping entry: map object
0x5105, subindex 1, 8 bits
  cop.obd(0x1A00,0x00) := 0x03;        // enable mapping by setting the
number of mapped objects
  cop.obd(0x1800,0x01) := dwCommParValue & 0x7FFFFFFF; // make 1st transmit
PDO valid (CiA: "Create TPDO"); clear bit 31

  print( "\r\nNew PDO mapping table:\r\n" );
  ShowPdoMap( cTPDO, 1/*PdoNumber*/ );  // show the new PDO mapping table
(function implemented in CANopen1.upt)
endproc; // TestPdoMapping()
```

script_demos\J1939sim.cvt

The script in this example simulates an ECU (electronic control unit) from which *a few* parameters can be read via J1939 protocoll.

script_demos\VT100Emu.cvt

This example emulates a VT100- or VT52-Terminal, using the simulated 'text screen' (text panel).
Works with CAN (up to 8 characters per CAN message) and the serial port (RS-232).
An overview of the most important VT100- and VT52-Escape-Sequences can be found on the "MKT-CD" (available 'online'), in Document Nr. 85141, VT100/VT52-Emulation für MKT-Geräte (so far only available in german language, but the Escape sequences should be easy to grasp).

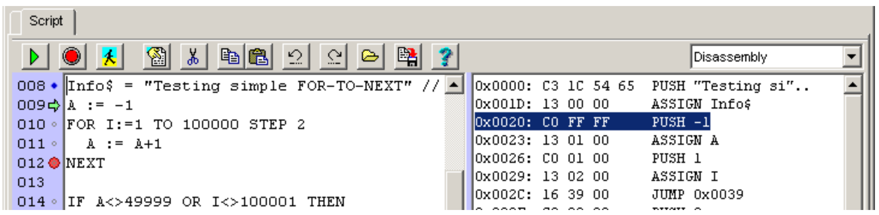
To load one of these examples into the programming tool, select 'File' .. 'Load Program' in the programming tool. Remember, any script is part of a display application (*.cvt or *.upt), it is not saved in an extra file.
You will find the examples in the tool's subdirectory 'programs\script_demos' (not to be confused with the windows 'Programs' / 'Program Files' / 'Programmi' folder .. there is no such language-dependent nonsense directory *inside* the programming tool). Sometimes the windoze file selector will enter that directory immediately. Otherwise, find your way to the directory where the ***last version of*** the programming tool has been installed on your PC.

5. Bytecode

Note:

This chapter is considered 'worth reading', but it is not essential to use the script language. It's up to you to read it, or ignore it. If you like to know what's going on 'under the hood', please proceed :o)

To increase the execution speed of the script, the sourcecode is translated from plain text ("ASCII") into a tokenized binary code (bytecode). For various reasons, the bytecode isn't the same as the microcontroller's native machine code (because in that case, the script couldn't be easily simulated and debugged in the programming tool). But the microcontroller can execute this code much faster than interpreting the sourcecode directly (unlike the 'display interpreter', which interprets event definitions and expressions 'directly', without RPN, and without tokenisation).



```
Info$ = "Testing simple FOR-TO-NEXT" //
A := -1
FOR I:=1 TO 100000 STEP 2
  A := A+1
NEXT

IF A<>49999 OR I<>100001 THEN
```

```
0x0000: C3 1C 54 65   PUSH "Testing si"..
0x001D: 13 00 00      ASSIGN Info$
0x0020: C0 FF FF      PUSH -1
0x0023: 13 01 00      ASSIGN A
0x0026: C0 01 00      PUSH 1
0x0029: 13 02 00      ASSIGN I
0x002C: 16 39 00      JUMP 0x0039
```

Screenshow of script editor (left) with disassembly (right)

Hint:

If you're interested, the bytecode can be seen in the 'disassembly view'. In the programming tool, open the script editor tab, click on the menu item in the editor's toolbar, and select 'Show Disassembly / Bytecode'.

1 5.1 Compiling the sourcecode into bytecode

The human-readable sourcecode will translated into machine-executable bytecode after loading a display program, so -as a user- you don't need to care about this. This translation process (called 'compilation') takes place in the programming tool as well as in the real target (firmware), because the bytecode running on the real target is not exactly the same as in the programming tool.

During compilation, numeric expressions ("formulas") are converted from the normal mathematic 'infix' notation into 'postfix' alias Reverse Polish Notation (RPN). You will hopefully never have to worry about this (at least not as long as the compiler can parse your code..). Here is an example for a tokenized statement ("A := 1 + A * (3-1) "). The first line contains the original sourcecode, the second line shows the RPN (which is almost the same as the bytecode in symbolic form) :

```
Sourcecode      : A := 1 + A * (3-1) ;
RPN / bytecode  : 1 A 3 1 - * + ASSIGN(A)
```

RPN is evaluated from left to right. Operands (constant values and variables) are pushed on the stack, while operators (like "+"="ADD" or "*"="MULTIPLY") usually pop two values from the stack, and push the result back on the stack. At the end of an RPN evaluation, the top of the stack contains the result. A list of bytecode operators can be found in the bytecode specification in one of the next chapters.

Hint:

If you are curious about RPN, and why it's so much easier to evaluate for a machine than the classic 'infix' notation, search the net for an article titled 'Postfix Notation Mini-Lecture' by Bob Brown. But again, to use the script language, you don't need to understand all the details.
The debugger's disassembly view also shows the bytecode, with one instruction per line. You can watch the execution of the code (especially the evaluation of RPN expressions) while single stepping, and see how intermediate values are pushed on, and popped from the stack. During single-step, the stack contents are displayed in the editor's status line.
The bytecode concept may sound similar as the one used in a Java Virtual Machine, but they are not compatible. Each 'value' on the stack carries it's data type along with the value, which is not the case in a Java VM.

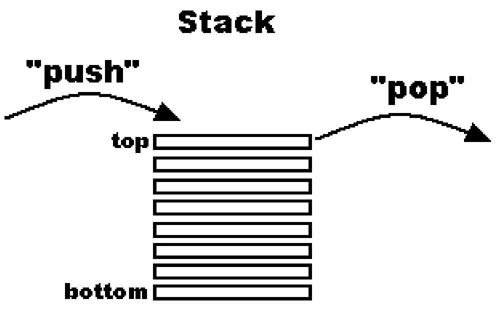
The amount of 'code memory' (RAM for the bytecode) is limited on the target system. As of 2010-08-03, the maximum size was 32767 bytes. This figure may vary, depending on the amount of RAM on the target system. The stack size is also limited to a few thousand entries.

To reduce the code memory requirements of your script program...

- Avoid unnecessary 'constant expressions' like this one:
  A := 1+2+3 // better use a calculated constant here !
- Use integer variables if you don't need floating point, last not least because the target CPU doesn't have a floating point unit (FPU).
  Beware that old BASIC interpreters used floating point if the data type isn't specified by the data type suffix .
  We don't : We use integer by default, because on the target system, integer calculations are much faster than floating point.
- Integer *constants* ranging from -32768 to +32767 require less code memory space than larger values, because there are different opcodes to push 'short' and 'long' values

5.2 The Stack

(Note for translators: Please don't translate "stack" into "Keller" in german - that's really misleading, even though frequently seen these days. In this context, "Stack" means "Stapel" in German, it hasn't got much to do with a german "Keller" = cellar or basement !).



The stack is a classic Last-In / First-Out buffer. If a value (or return address, or similar) is 'pushed' to the stack, it gets .. well .. **STACKED** (not "**cellared**") on top of the other stack elements. The topmost element on the stack can be 'popped' off the stack, which means read and remove it from the stack. These are the only operations performed on a basic stack !

To inspect the stack while debugging your code (in the programming tool), use the stack display . Keep an eye on the stack usage, especially if you program uses a lot of user-defined procedures or functions, with local variables and recursive calls !

In the script language, the stack is used to evaluate RPN expressions, to store return addresses (during function- and procedure calls), and local variables. Local variables are allocated on the stack by pushing a dummy value (usually zero) to the stack, and saving the address of the first local variable in a register frequently called the 'base pointer'. The local variabes of the currently active function or procedure are stored in a so-called stack frame, which is explained in the next chapter.

1.1 5.2.3 Stack Frames (for function arguments and local variables)

As mentioned in the previous chapter, the stack is also used as a storage for local variables. A special register, called the 'Base Pointer' (BP, similar purpose as in the 8086 CPU from which the name was "borrowed"), points to the base of the stack frame of the currently active function. Through the base pointer, that part of the stack area ('stack frame') is accessed like a random-access memory using the base pointer plus an offset. For example, the bytecode instruction PUSH BP[2] reads the value of the BP register, adds an offset of two, reads a value from that address, and pushes it on top of the stack. In effect, it copies some value from one stack location to another, and increments the stack pointer. In constrast to the *stack pointer* (SP) which always points to the top of the stack, the *base pointer* (BP) remains constant within an instance of a user-defined function or procedure.

Here is an example for the *stack frame* inside a function with a few local variables (positive offsets), and two function arguments (negative base pointer offsets). Note that the stack entries with negative offset for the base pointer don't strictly belong to the callee's (called function's) local stack frame, but for simplicity, they can be accessed through the BP.
BP[n] means "the n-th element addressed through the base pointer", treating the stack frame like an array for simplicity.

```
BP[2] = third local variable
BP[1] = second local variable
BP[0] = first local variable (allocated" by the callee)
BP[-1] = old base pointer saved on the stack, pushed by callee
BP[-2] = return address on the stack, pushed by caller
BP[-3] = Number of arguments passed from caller to callee
BP[-4] = Last function argument (pushed last by the caller)
BP[-5] = First function argument (pushed first by the caller)
BP[-6] = Function result aka 'return value' .
 // Space for the 'return value' is reserved by the caller(!)
 // by pushing a zero; regardless of the return data type.
 // Because the 'Function result' remains on the stack
 // after cleaning up the argument list, the return value
 // is pushed first / popped last .
```

In contrast to most built-in functions (runtime library functions), user-defined functions don't remove the function arguments. Here, the caller removes those arguments from the stack which 'he' has pushed before the call, and the callee only cleans up those stack entries which 'he' has pushed there (like local variables, etc).

2 5.3 Bytecode specification (*very* preliminary !!!)

You only need to know what these bytecode instructions do, if you need to single-step through the assembly view. For 'normal' (source-level) debugging, skip this chapter - you will rarely need it.
Note that most of the binary operators (i.e. operators with two inputs) know the data type of the operands, because the stack contains the data type as well as the 'value' (each stack entry occupies EIGHT bytes for that reason; strings are a special breed... they are in fact "pointers" to a special string memory.

Mnemonic	Hex Opcode	Remarks
PUSH <value>	various types 0xC0..0xC6, ..	Pushes a constant, or a variable, to the stack. Data type is encoded in the lower bits of the opcode. Constants may be integer, float, or string; the value follows immediately after the opcode in code memory.
PUSH BP[n]	various (depending on n)	Pushes the value of a local variable, function argument, or similar element to the top of the stack. BP is the (virtual) base-pointer register. 'n' is an offset added to the base pointer value. Usually, positive offsets indicates a local variable, negative offsets are function arguments or similar.
POP BP[n]	various (depending on n)	Pops one element off the top of the stack, and copies it to a local variabe or similar (see PUSH BP..) .
ASSIGN <var>	various (type-dependent)	Pops one element off the top of the stack, and assigns it to the specified variable. Global variables are identified by a unique, non-negative, 8- or 16-bit "reference number".
RD <source>	various (type-dependent)	Read-access for arrays, and/or components of various structures. Depending on the complexity of the 'source', additional values (like array indices) may be popped off the stack. The read result will be pushed to the stack (similar as 'PUSH').
WR <dest>	various (type-dependent)	Write-access for arrays, and/or components of various structures. Depending on the complexity of the 'destination', additional values (like array indices) may be popped off the stack. The to-be-written value will also be popped off the stack.
ADD	0x20	pops two elements off the stack, adds them, and pushes the result on the stack
SUBTRACT	0x21	pops two elements off the stack, subtracts them, and pushes the result on the stack
MULTIPLY	0x22	pops two elements off the stack, multiplies them, and pushes the result on the stack . The resulting data type depends in the operands: If both operands are integer, the result is an integer. If any of the operands is a floating point value, the result will be a

		floating point value, too.
DIVIDE	0x23	pops two elements off the stack, divides them, and pushes the result on the stack . Note: If both inputs are INTEGERS, the result will also be an INTEGER value (not floating point) !
MODULO	0x24	pops two elements off the stack, calculates their division remainder, and pushes the result on the stack
COMP >=	0x25	pops two elements off the stack, compares them for "greater or equal", and pushes the boolean result (zero for FALSE, one for TRUE) on the stack.
COMP <=	0x26	pops two elements off the stack, compares them for "less or equal", and pushes the boolean result (zero for FALSE, one for TRUE) on the stack.
COMP ==	0x27	pops two elements off the stack, compares them for "equality", and pushes the boolean result (zero for FALSE, one for TRUE) on the stack.
COMP <>	0x28	pops two elements off the stack, compares them for "non-equality", and pushes the boolean result (zero for FALSE, one for TRUE) on the stack.
COMP >	0x29	pops two elements off the stack, compares them for "greater than", and pushes the boolean result (zero for FALSE, one for TRUE) on the stack.
COMP <	0x2A	pops two elements off the stack, compares them for "greater than", and pushes the boolean result (zero for FALSE, one for TRUE) on the stack.
AND	0x2B	pops two elements off the stack, AND-combines them (boolean), and pushes the result on the stack
OR	0x2C	pops two elements off the stack, OR-combines them (boolean), and pushes the result on the stack
BIT_AND	0x2D	pops two elements off the stack, AND-combines them (bitwise), and pushes the result on the stack
BIT_OR	0x2E	pops two elements off the stack, OR-combines them (bitwise), and pushes the result on the stack
	..	
	..	

U_MINUS	0x1D	Inverts the sign of the element on top of the stack, for example turns 1.234 into -1.234 .
NOT	0x1E	Turns the element on top of the stack into it's (one's) complement, for example, zero (boolean "FALSE") is turned into one (boolean "TRUE") and vice versa .
	..	
JUMP	0x04, 0x05	absolute jump to the specified 'short' or 'long' address
POPJZ	0x06	POP and Jump if Zero (to the specified address) Pops one (hopefully numeric) value off the stack, and jumps to the specified address if the value is ZERO . This instruction is frequently emitted by the compiler to produce IF-THEN-ELSE, and REPEAT-UNTIL constructs .
	..	
PROC	0x70 (?)	Begin of a user-defined procedure. The operand field contains information about the procedure's stack frame, and the offset from the function's start and end (so it can easily be skipped in memory).
ENDPROC	0x71 (?)	End of a user-defined procedure. Cleans up the procedure's stack frame, and returns to the caller.
CALL PROC	0x72 (?)	Calls (invokes) a user-defined procedure. Expects NO return-value.
FUNC	0x73 (?)	Begin of a user-defined procedure. Similar to a procedure, but a function returns a value.
ENDFUNC	0x74 (?)	End of a user-defined function. Cleans up the function's stack frame, and returns to the caller. Leaves the 'result' (special location on the stack) unchanged.
RETFUNC	0x75 (?)	Pops one value off the stack, copies it into the function's *return value*, cleans up the function's stack frame, and returns to the caller. After that (after returning), the caller will find the return value on the 'new' top of the stack.
CALL FUNC	0x76 (?)	Calls (invokes) a user-defined function. In contrast to a procedure-call, expects a return value. After returning from callee to the caller, the function's return value ("result") remains on the RPN stack.
	..	
	..	
	..	
	..	

Note: There may be many opcodes missing in the above table, which have not been documented here due to a lack of time of the software development engineer ;-)

See also:

Overview (link to an external file, only works in the online help (html), not in the PDF variant of this file)